NORTHWESTERN UNIVERSITY

Methods for Images, Time Series, and Activation Functions with Deep Neural Networks

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Engineering Sciences and Applied Mathematics

By

Mark Harmon

EVANSTON, ILLINOIS

September 2018

ABSTRACT

Methods for Images, Time Series, and Activation Functions with Deep Neural Networks

Mark Harmon

In this work, we explore the utility of the three main types of neural networks: feed forward, convolutional, and recurrent. While using these networks, we develop a new way to model multiagent trajectory data, explore the use of multiple activation functions for neurons at each layer of a neural network, and create a new model that makes a dynamic number of predictions with sequence data.

Acknowledgements

First and foremost, I want to thank my advisor Diego Klabjan for his mentorship and guidance during my years at Northwestern. He taught me to be both passionate and thorough with my research. I appreciate all of time and effort he has given me. I also appreciate all of the ideas and the funding provided to complete my Ph.D. I am very thankful to have such a great advisor.

I want to thank my other committee members: David Chopp and Alvin Bayliss for their continued support and time. Both David and Alvin were excellent teachers and mentors during my first year of studies at Northwestern University, and I really appreciate everything they taught me.

I also want the thank all the other professors who taught me classes in the Department of Engineering Sciences and Applied Mathematics. My first year at Northwestern University was one of the most influential years of my life, and I can't thank the professors enough for how much I learned during that period of time.

I also want to thank Patrick Lucey for his guidance at STATS LLC. I enjoyed a great work environment under his team and enjoyed working with him on the paper that became Chapter 2 of my thesis. He taught me how to mix both research and business to create a viable product.

Last but not least, I want to thank all of my friends and family for being supportive during my studies. I especially would like to thank my father and mother who try to understand the work I'm doing and encouraging me throughout my years at Northwestern. I want to extend a

special thank you to all the friends I've made over the years here. Being able to research while surrounded by a great group of people was amazing.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

The goal of this study is to create valuable new methodologies for solving problems with deep neural networks. This work includes the three main types of neural networks: feed forward (FFN), convolutional (CNN), and recurrent (RNN). Deep neural networks have a range of uses and have been successful in applications such as: image recognition, foreign language translation, image generation, and video classification.

These models are becoming more widely adopted in industrial settings by technological giants. Services such as Google Voice and Alexa rely on these powerful machine learning tools to create products we use daily. Since this field is finding success in a variety of data settings, pushing the limits of these powerful models is imperative. This work pushes the boundaries of a variety of neural network architectures and pushes the scope of the types of problems that can be solved by deep neural networks. Before we begin with the main studies of this work, we first introduce the primary components of deep neural networks, also known as "deep learning." In the following chapter, we explore feed forward, convolutional, and recurrent neural networks.

1.1. Feed Forward Neural Networks

Neural networks are effective because of their ability to extract features from raw data. In typical machine learning models, we must construct features from the raw data for our model inputs. Hand-crafting features is time consuming and unreliable. It may take a long time to find features that explains the variance of the dataset. A model that can extract features from

raw data can save a lot of time otherwise spend creating features. The model may also improve accuracy since it can automatically discover important characteristics that are missed when manually hand-crafting features.

Feed Forward Networks (FFN) are the most basic form of a deep neural network. The architecture of an FFN consists of a number of layers ℓ, weights θ, and activation functions a. The goal of an FFN is to find the weights θ that maximize the likelihood that the network represents the relationship between the data X and the labels of the data Y. During the training phase, the error between the outputs of the network $f(X;\theta)$ and the labels Y is used to calculate the next gradient step. For a classification problem, log loss is the function used to calculate the error. At each layer ℓ, there are a specified number of neurons that have a specified activation function a^{ℓ}. The equations for a typical FFN with the Kullback-Leibler divergence as the loss function are as follows:

$$\min_{\theta} E_{(X,Y)} KL(Y||f(X,;\theta)) = E_{(X,Y)} \sum_{i} y_i \log f_i(X;\theta) \quad \text{Optimization Function}$$

$$z^0 = X \quad \text{inputs of the model}$$

$$z_i^{\ell}(X;\theta) = a^{\ell}\Big(\sum_{k=1}^{K} z_k^{\ell-1}\theta_{ik}^{\ell-1}\Big) \quad \text{for neuron } i \text{ in layer } \ell \text{ and } K \text{ neurons at layer } \ell-1$$

$$f_i(X;\theta) = \frac{e^{z_i^L \theta_i^{L^T}}}{\sum_m e^{z_m^L \theta_i^{L^T}}} \quad \text{softmax function for probability of each class } i$$

To clarify the FFN architecture, we provide a visual representation of a feed forward neural network with Figure 1.1. The data moves through the model from the bottom to the top. Each neuron at the input layer represents a feature from the data to be run through the model. Each

input neuron connects to every neuron in the first hidden layer via a trainable weight, which are typically represented as θ. At each hidden layer, there are a specified number of neurons, each with an activation function a. Typically, the activation functions are the same for each layer, but we challenge this notion later in Chapter 3. In the Figure 1.1, the number of neurons is the same. This does not need to be the case. Typically, the number of neurons are treated as hyperparameters to find the best possible architecture depending on the dataset and model depth.

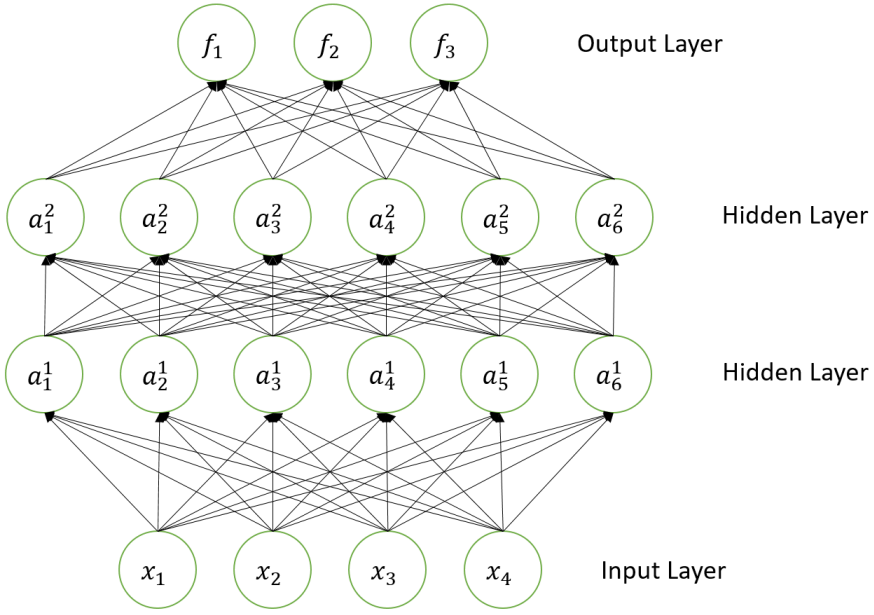

Figure 1.1. Visual representation of a Feed Forward Network (FFN) with two hidden layers with six neurons each and three outputs. Note that it is not necessary for each hidden layer to have the same number of neurons.

Usually we favor model depth over layer width due to the former yielding higher accuracy on most datasets. In other words a model with more layers and less neurons tends to perform better than a shallow network with many neurons. The current knowledge for this discrepancy is that increased depth allows the model to learn more advanced features at each layer. Another

way of viewing this phenomena is by comparing a FFN to a logistic regression model. Both models end in a log loss function for classification problems. The difference is that a FFN transforms the input data at each hidden layer to another space before classification. At each transformation, the network transfers the data to a space that is more easily separable. Therefore, with more hidden layers, the data can easily be separated at the end of the network, which is essentially a logistic regression model.

A feed forward network has the name "feed forward" because of the two-step process for model training. First, the data inputs X are fed into the model, and each layer calculates a linear combination of the weights and inputs followed by an activation function. At the network's output layer, the output probabilities are calculated by the softmax function. Then the log loss between the outputs and true labels is measured. This is the feed-forward step.

The next step for training a FFN is the backpropagation step. After the error calculation, we compute the gradient of the network with respect to the weights θ. The derivative calculations are the most time-consuming portion of training a network. The derivatives are calculated through the chain rule, but the large number of computations can bog down the training phase. This is an important consideration when choosing the activation functions for a network. It is imperative to have an activation function that trains quickly and effectively. There are a number of typical choices for a FFN. A small fraction of these choices are:

- Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$
- Tanh: $f(x) = \tanh(x)$
- ReLU: $f(x) = \max(x, 0)$

All of these activations have derivatives that are easy to calculate. More importantly, these functions do not have exploding gradients, which may cause enormous issues when backpropagating through many hidden layers. Sigmoid functions were by far the most popular activation function before neural networks gained traction in the late 2000's. At the time, computational power was not widely available. Computers lacked both memory and cheap parallel computing capabilities. Also, the gradient of the sigmoid function has a maximum value of 0.25. This leads to an additional difficulty in training deep networks since the gradient becomes arbitrarily small near the input layer of a network. Therefore, for most architectures, the ReLU (Rectified Linear Unit) is used.

The ReLU was first used by Nair and Hinton (2010) for a type of network called a Restricted Boltzmann Machine (RBM). We are not reviewing RBM's in our work here, but ReLU's broke the computation barrier for training deep neural networks. The derivative is simple, being 1 or 0 except for the case in which $x = 0$. The ReLU is advantageous because of its gradient does not vanish during backpropagation with deep networks. Its behavior also resembles the spiking activation of a neuron seen in brain cell models.

1.2. Convolutional Neural Networks

In this section, we explore another main architecture used for neural networks, convolutional neural networks (CNN's). CNN's were made specifically for a difficult ImageNet dataset by Krizhevsky, Sutskever, and Hinton (2012). The original version of the dataset contained 200,000 training images with 1000 categories/labels. The images contained pictures of a variety of objects to classify including: vehicles, pets, and other wildlife.

FFN's are effective for a variety of problems, but they struggle when presented with image classification. A FFN treats each pixel of information from an image as i.i.d., which causes numerous issues for image classification. To clarify the weaknesses of a FFN, we utilize the MNIST dataset, which consists of black and white images of hand-written digits as seen in Figure 1.2.

Figure 1.2. A few images from MNIST training set. Each is composed of 28x28 pixels.

An FFN can solve this dataset with over 80% in its original form. However, an FFN will mis-classify most examples if the digits in the images are moved one pixel to the left or right. The images remain the same to the observer's eye. Therefore, the FFN network is over-fitting and more important missing the context of the relationship of one pixel to another. To prevent over-fitting, we can shift the images an arbitrary number of pixels to the left or right, flip the images, rotate the images ϕ^o, and stretch the images. Introducing noise into the training data is effective for the MNIST dataset, but it is inadequate for more complicated datasets such as ImageNet. The ImageNet images each are $3 \times 256 \times 256$ (3 extra dimensions for RGB), which is too complex for the standard FFN.

CNN's are derived from the use of filters in computer vision. Filters are used to extract features from an image. Some filters in computer vision may extract the edges of an image while other filters may add Gaussian noise to make an image "fuzzy." Filters are effective at feature extraction for images, but we do not want to be encumbered with hand-crafting features.

Therefore, rather than using known filters to extract information from a network, a convolutional neural network learns the weights θ for a variety of filters.

A CNN is composed of two parts. One is the convolutional layer and the other is a pooling layer. At the output layer, a CNN usually connects to a feed forward layer followed by a softmax function. In Figure 1.3, we present the architecture of a typical CNN. It is composed of only a single convolution and pooling layer before connected to a FFN hidden layer. Like a FFN, a CNN can contain any number of layers, any number of filters, and any number of pooling layers.

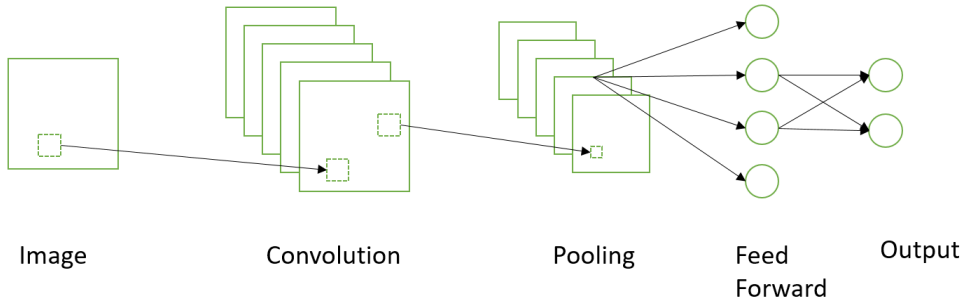

Figure 1.3. Visual representation of a Convolutional Neural Network (CNN). This example has four filters and a single convolution and pooling layer. Note that a CNN can have a number of convolution and pooling layers, and it has been shown that having no pooling layer at all can result in higher accuracy Springenberg et al. (2014).

To understand CNN's, we explain the two main components: the convolution layer and the pooling layer. The convolution layer contains two hyperparameters: the filter sizes and the number of filters (number of neurons).

The shape of a filter can change depending on the type of problem being solved, but the shape used for most image problems is square. This is subject to change depending on the behavior of the dataset. For example, in Chapter 4 we use 1D convolutions to extract information

from a multiple time series signal. The filter size determines how much context the model learns from one pixel to the next. Rather than treating each pixel as i.i.d. as in a FFN, the filter learns the relationship between pixels that are close to one another.

Under the extreme case of a 1×1 filter κ, a convolutional layer is computationally equivalent to a feed forward layer. It is important to note that each $n \times m$ filter is used across the entire input image. Therefore, a typical CNN will have far fewer trainable parameters than a FFN, which helps prevent over-fitting. The convolutional step at layer ℓ with an $n \times m$ filter is governed by the following equation:

$$x_{ij}^{\ell} = \sum_{a=0}^{n-1} \sum_{b=0}^{m-1} \kappa_{ab} a_{(i+a)(j+b)}^{\ell-1}$$

The next part of a CNN is the pooling layer. By far the most common pooling layer is the max pooling layer. After the convolutional layer, a max-pooling layer is typically used to reduce the number of features and extract the most important features from the convolution step. A pooling layer reduces the size of each filtered image, but it does not traditionally change the number of filtered images. Each pooling layer has a predetermined pooling size, which like a filter size, determines the number of pixels to consider in relationship with one another. A max pooling layer finds the maximum value given the pooling size for $n \times m$ pixels,which moves across the entire filtered image. It is important to note that recent work by Springenberg et al. (2014) shows that using only convolutional layers may lead to better training and generalization. In Chapters 3 and 4, we utilize only convolution layers and no pooling layers for this reason.

1.3. Recurrent Neural Networks

One of the most relevant problems for large datasets is time series prediction. Some example problems in this space are sentiment analysis of written documents, language translation, chat bots, and financial security prediction. After residual networks by He, Zhang, Ren, and Sun (2016) and inception networks by Szegedy, Liu, Jia, Sermanet, Reed, Anguelov, Erhan, Vanhoucke, and Rabinovich (2015) recently garnered amazing results for image-based problems, deep learning interests shifted toward sequence problems.

FFN's struggle with sequences because they fail to identify relationship between each input of a time series. A recurrent neural network (RNN) takes an input sequence one at a time and remembers each previous input. An RNN remembers previous inputs by creating a hidden vector that contains the features of previous inputs and combines this vector with the current input. In Figure 1.4 we present a graphical representation of an RNN.

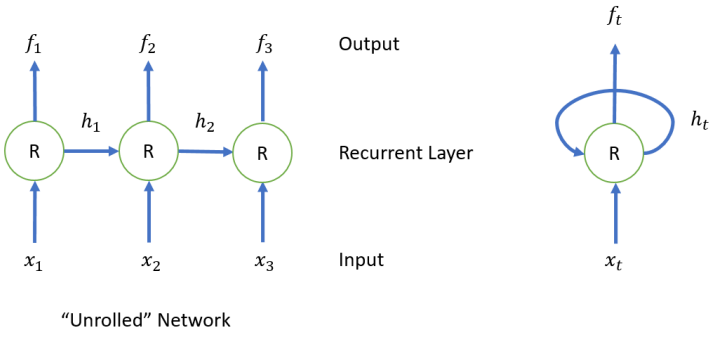

Figure 1.4. Visual representation of a Recurrent Neural Network (RNN). On the right is the typical structure of an RNN and the left is an "unrolled" version showing memory extraction with h_1 and h_2 along with each individual input into the network.

In Figure 1.4, we show the typical structure of an RNN on the right and the "unrolled" version of the same network on the left. The "R" node represents a recurrent layer. Similar to

other neural networks, RNN's may have any number of layers. At each layer n of an RNN, the hidden state h_t^n transfers to layer n only at time $t+1$ (not any other layer i receives the hidden vector from layer n). In addition, a recurrent layer contains separate weight matrices W_i where i can refer to the output vector f, hidden vector h, and input x. Note that there are versions of RNN's that pass hidden vectors of layer n to different layers k of the network, but we do not use these variants in our study. The following are the basic equations for a simple recurrent layer at time t:

$$f_t = W_f h_t$$

$$h_t = \sigma(W_h h_{t-1} + W_x x_t)$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

The forward step of an RNN is very similar to that of an FFN. The only difference is that class evaluations are done at each time t rather than a single time with all current features. This small difference in the forward step causes a major difference with the backpropagation step. Since the hidden vector h_t is dependent upon each previous hidden vector $h_{i<t}$, the chain rule must be applied backwards in time as well as in layers. Therefore, each sequence input is essentially adding a layer to the model when considering backpropagation. The computational complexity compounded when a recurrent model consists of multiple recurrent layers. When constructing a RNN, one must carefully consider model depth and sequence input length.

In Section 1.1, we stated that sigmoid functions lead to low performance and ReLU functions are the preferred activation function for neural networks. However, since RNN's backpropagate through time, there is a danger of exploding gradients. The sigmoid function prevents

exploding gradients, but it causes vanishing gradients for large sequences or deep networks. Thus, Hochreiter and Schmidhuber (1997) present a layer they dub the Long Short Term Memory (LSTM) layer. These are the layers that we use for all of our RNN's, which are in Chapter 4. In their paper, they show that their new LSTM layer has an effective gradient of 1, which prevents the vanishing gradient problem present in basic recurrent networks. The following are the basic equations of an LSTM layer with trainable weights W_{nm}:

$$f_t = \sigma(W_{fh}h_{t-1} + W_{fx}x_t)$$

$$i_t = \sigma(W_{if}h_{t-1} + W_{ix}x_t)$$

$$\widetilde{c_t} = \tanh(W_{ch}h_{t-1} + W_{cx}x_t)$$

$$c_t = f_t C_{t-1} + i_t \widetilde{c_t}$$

$$o_t = \sigma(W_{oh}h_{t-1} + W_{ox}x_t)$$

$$h_t = o_t \tanh(C_t)$$

The main idea of an LSTM layer centers around the cell state c_t. When an input x_t is entered into the model, the layer first calculates the forget gate f_t to determine how much of the old cell state to forget. After calculating the proposed cell state $\widetilde{c_t}$, the new cell state c_t is computed followed by the output values h_t. Note that even though this structure is more complicated than a simple RNN, the computational overhead is minimal in our experience.

1.4. Regularization

A typical neural network usually includes an additional term in the loss function for regularization, which we present here as $\Omega(\theta)$ with hyperparameter μ. Here, a hyperparameter refers to a parameter than is trained vida cross-validation instead of by gradient descent.

$$\min_{\theta} E_{(X,Y)} KL(Y||f(X,;\theta)) + \mu\Omega(\theta)$$

Because of the extensive number of trainable parameters in a neural network, there is a lot of potential for over-fitting. Typical functions for $\Omega(\theta)$ are the L1 and L2 norms which cause weight sparsity and lower weight magnitude respectively. In addition to a regularization function, neural networks are typically regularized using a leave-one-out validation data set. The validation set is not used during training, but is tested at each epoch, which consists of training over all examples in the training set once. After the loss of the validation set begins to increase drastically or the accuracy begins to decrease within some tolerance, we halt training.

Both L1 and L2 norms are used for most neural network applications. Overall in our work, we found that the L2 norm tends to produce better results. However, there is a more popular version of regularization that we use in nearly all networks, especially our deep architectures, called batch normalization created by Ioffe and Szegedy (2015). We found that batch normalization both decreases over-fitting and increases the training speed of the network.

Batch normalization renormalizes the data via z-score at each layer of a network to prevent stochastic drift from one layer to the next. During the training phase, it calculates the mean and variance of the incoming activations from the previous layer of the network. The means and standard deviations are used to normalize the activation for each batch of data during training.

Note that this only works when training using a batch gradient method, but this is the standard practice in neural networks.

At the beginning of the training phase, the batch normalization layer contains a running mean and running standard deviation. These are both used during the prediction phase of a neural network to normalize the validation and test set inputs at each layer.

CHAPTER 2

Image Analysis in the NBA

2.1. Contribution

In this paper, we predict the likelihood of a player making a shot in basketball from multiagent trajectories. Previous approaches to similar problems center on hand-crafting features to capture domain specific knowledge. Although intuitive, recent work in deep learning has shown this approach is prone to missing important predictive features. To circumvent this issue, we present a convolutional neural network (CNN) approach where we initially represent the multiagent behavior as an image. To encode the adversarial nature of basketball, we use a multi-channel image which we then feed into a CNN. Additionally, to capture the temporal aspect of the trajectories we use "fading." We find that this approach is superior to a traditional FFN model. By using gradient ascent, we were able to discover what the CNN filters look for during training. Last, we find that a combined FFN+CNN is the best performing network with an error rate of 39%.

2.2. Introduction to Prediction with Player Trajectories

Neural networks have been successfully implemented in a plethora of prediction tasks ranging from speech interpretation to facial recognition. Because of ground-breaking work in optimization techniques (such as batch normalization, Ioffe and Szegedy (2015)) and model architecture (convolutional, deep belief, and LSTM networks), it is now tractable to use deep neural networks to effectively learn a better feature representation compared to hand-crafted methods.

One area where such methods have not been utilized is the space of adversarial multiagent systems (for example, multiple independent players in competition), specifically when the multiagent behavior comes in the form of trajectories. There are two reasons for this: i) procuring large volumes of data where deep methods are effective is difficult to obtain, and ii) forming an initial representation of the raw trajectories so that deep neural networks are effective is challenging. In this paper, we explore the effectiveness of deep neural networks on a large volume of basketball tracking data, which contains the x, y locations of multiple agents (players) in an adversarial domain (game).

To thoroughly explore this problem, we focus on the following task: "given the trajectories of the players and ball in the previous five seconds, can we accurately predict the likelihood that a player with position/role X will make the shot?" For this paper, player role refers to a more fluid position of a player, which was explored by Lucey, Bialkowski, Carr, Morgan, Matthews, and Sheikh (2013). For example, a player may not be in the point guard position during the entire play. Since we plan to utilize an image representation for player trajectories, we use a convolutional neural network (CNN), which is widely considered a powerful method for image classification problems.

In this study, we treat each player as a generic player, i.e. we are not using player identities. By modeling generic players rather than individuals, we can more easily quantify the differences between an individual player and a generic player. Consider, for example, a rookie (which clearly has limited historical data). After a few games, a coach can compare his shooting performance against the performance of a generic player, which can be obtained by our model. Our model offers such comparisons at a very fine granular level of play. The coach can then guide the player to improvements. The same argument is applicable in other cases such as a

player getting more play-time (the model allows comparison against generic players in addition to his past performance, which clearly does not need our model). To obtain these values, we want to identify shooting for each generic position (point guard, shooting guard, center, power forward, and small forward). Since we classify whether the shot will be made for a single offensive position, every offensive player corresponds to either the class of making a shot or missing a shot. Therefore, our classification problem consists of ten classes.

Our work contains three main contributions. First, we represent trajectories for the offense, ball, and defense as an eleven channel image. Each channel corresponds to the five offensive and defensive players, as well as the ball. To encode the direction of the trajectories, we fade the paths of the ball and players. In our case, an instance is a possession that results in a shot attempt. Second, we apply a combined convolutional neural network (CNN) and feed forward network (FFN) model on an adversarial multiagent trajectory based prediction problem. Third, we gain insight into the nature of shot positions and the importance of certain features in predicting whether a shot will result in a basket.

Our results show that it is possible to solve this problem with relative significance. The best performing model, the CNN+FFN model, obtains an error rate of 39%. In addition, it can accurately create heat maps by shot location for each player role. Features which are unsurprisingly important are the number of defenders around the shooter and location of the ball at the time of the shot. In addition, we found that our image-based model performs just as well as a feed-forward technique that required us to build nearly 200 features.

2.3. Review of Multiagent Systems in Sports Prediction

With the rise of deep neural networks, sports prediction experts have new tools for analyzing players, match-ups, and team strategy in these adversarial multiagent systems. Trajectory data was not available at the time, so much previous work on basketball data using neural networks have used statistical features such as: the number of games won and the number of points scored. For example, Loeffelholz, Bednar, Bauer et al. (2009), use statistics from 620 NBA games and a neural network to predict the winner of a game. Another interested in predicting game outcomes is McCabe and Trevathan (2008). On the other hand, Nalisnick (2014) in his blog discusses predicting basketball shots based upon the type of shot (layups versus free throws and three-point shots) and where the ball was shot from. In other sports related papers, Huang and Chang (2010) use a neural network to predict winners of soccer games in the 2006 World Cup. Also, Wickramaratna, Chen, Chen, and Shyu (2005) predict goal events in video footage of soccer games.

Although aforementioned basketball work did not have access to raw trajectory data, Lucey, Bialkowski, Carr, Yue, and Matthews (2014), use the same dataset provided by STATS for some of their work involving basketball. They explore how to get an open shot in basketball using trajectory data to find that the number of times defensive players swapped roles/positions was predictive of scoring. However, they explore open versus pressured shots (rather than shot making prediction), do not represent the data as an image, and do not implement neural networks for their findings. Other trajectory work includes using Conditional Random Fields to predict ball ownership from only player positions (Wei, Sha, Lucey, Carr, Sridharan, and Matthews (2015)), as well as predicting the next action of the ball owner via pass, shot, or dribble Yue, Lucey, Carr, Bialkowski, and Matthews (2014). Miller, Bornn, Adams, and Goldsberry (2014)

use non-negative matrix factorization to identify different types of shooters using trajectory data. Because of a lack of defensive statistics in the sport, A. Franks et. al. create counterpoints (defensive points) to better quantify defensive plays (Franks, Miller, Bornn, Goldsberry et al. (2015b) Franks, Miller, Bornn, and Goldsberry (2015a)). Perše, Kristan, Kovačič, Vučkovič, and Perš (2009) make use of trajectory data by segmenting a game of basketball into phases (offense, defense, and time-outs) to then analyze team behavior during these phases.

Wang and Zemel (2016) use trajectory data representations and recurrent neural networks (rather than CNN's) to predict plays. Because of the nature of our problem, predicting shot making at the time of the shot, there is not an obvious choice of labeling to use for a recurrent network. They also fade the trajectories as the players move through time. Like us, they create images of the trajectory data of the players on the court. Our images differ in that we train our network on the image of a five second play and entire possession, while their training set is based on individual frames represented as individual positions rather than full trajectories. They use the standard RGB channels, which we found is not as effective as mapping eleven channels to player roles and the ball for our proposed classification problem. Also, the images they create solely concentrate on the offense and do not include defensive positions.

In addition, work by Cervone, D'Amour, Bornn, and Goldsberry (2016) also explores shots made in basketball using the same STATS data. Our work focuses on a representation capturing the average adversarial multiagent behavior, compared to the approach of Cervone et.al., which focuses on individual players. In addition, while they concentrate on a Markov model that transitions between a coarse and fine data representation, our goal is to represent the fine-grained data for a deep learning model. Our work focuses on shot prediction for the average player in the NBA at the time of the shot while their work formulates a way to calculate estimated point

value based upon specific player identity without reporting predictions for individual plays. Their estimated point value includes the probability of making a shot; however, this probability is based on individual characteristics of a player prior to taking the shot which is different from our goal of considering generic players. We do not see an easy way to modify their models to provide answers based on our setting. In summary, there is no readily available numerical comparison of the two models that would provide answers for shot making predictions of generic players.

The final model that we implement, the combined network, utilizes both image and other statistical features. There is work that utilizes both image and text data with a combined model. Recently, Xu, Ba, Kiros, Cho, Courville, Salakhutdinov, Zemel, and Bengio (2015), Karpathy and Fei-Fei (2015), Socher, Karpathy, Le, Manning, and Ng (2014), Vinyals, Toshev, Bengio, and Erhan (2015b), and Mao, Xu, Yang, Wang, Huang, and Yuille (2015) all explore the idea of captioning images, which requires the use of generative models for text and a model for recognizing images. However, to the best of our knowledge, we have not seen visual data that incorporates fading an entire trajectory for use in a CNN.

2.4. STATS LLC NBA Trajectory Data

The dataset was collected via SportsVU by STATS LLC. SportsVU is a tracking system that uses 6 cameras to track all player locations (including referees and the ball). The data used in this study was from the 2012-2013 NBA season and includes thirteen teams, which have approximately forty games each. Each game consists of at least four quarters, with a few containing overtime periods.

Table 2.1. Trajectory Data Sample

Game Time	Real Time	Team	Player	X	Y	Z	Role	Event
693	514200	1	101	21.5	33.6	0.0	1	0
693	514200	1	102	24.1	14.1	0.0	2	1
693	514200	1	103	5.4	9.6	0.0	3	0
693	514200	1	104	3.9	45.6	0.0	4	0
693	514200	1	105	10.4	3.5	0.0	5	0
693	514200	2	201	13.6	31.6	0.0	1	0
693	514200	2	202	20.4	15.4	0.0	2	0
693	514200	2	203	7.7	13.3	0.0	3	0
693	514200	2	204	6.0	38.6	0.0	4	0
693	514200	2	205	13.9	13.2	0.0	5	0
693	514200	-1	-1	25.1	14.0	3.4	0	0
693	514200	-2	1	16.9	49.2	0.0	0	0
693	514200	-2	3	78.9	0.5	0.0	0	0
693	514200	-2	2	26.0	3.1	0.0	0	0

The SportVU system records the positions of the players, ball, and referees 25 times per second. At each recorded frame, the data contains the game time, the absolute time, player and team anonymized identification numbers, the location of all players given as (x, y) coordinates, the role of the player, and some event data (i.e. passes, shots made/missed, etc.). It also contains referee positions, which are unimportant for this study, and the three-dimensional ball location. Table 2.1 shows a sample (with player identities masked) of a single frame of data.

The sample detailed above shows a single frame snapshot from a game where team 1 is playing team 2. ”Game Time” refers to time left in the quarter in seconds while ”Real Time” is the actual time of the day outside of the game. The next column is the team labels where the ball and referees are denoted with a ”-1” and ”-2,” respectively. Each player has an ID along with the ball and each referee. Next are the coordinates of all players, referees, and the ball along with the role/position of the player and special event codes for passes, shots, fouls, and rebounds.

This dataset is unique in that before SportVU, there was very little data available of player movements on the court and none known that provides frame-by-frame player locations. Since it is likely that most events in basketball can be determined by the movements of the players and the ball, having the trajectory data along with the event data should provide a powerful mixture of data types for prediction tasks.

There are a few ways to extract typical shot plays from the raw data. One is to choose a flat amount of time for a shooting possession. In our case we choose to include the final five seconds of a typical possession. To obtain clean plays, those that lasted less than 5 seconds due to possession changes and those in which multiple shots were taken were thrown out. After throwing out these cases, we were left with 75,000 five second plays.

The other way of obtaining play data would be to take the entire possession. Thus, rather than having plays be limited to five seconds, possessions can be much longer or shorter. Since the raw data does not contain labels for possession, we had to do this ourselves. To identify possession, we calculate the distance between the ball and each of the players. The player closest to the ball would be deemed the ball possessor. Since this approach may break during passes and other events during a game, we end possession when the ball is closer to the defensive team for 12 frames (roughly 0.5 seconds). The procedure yields 72,000 possession examples. We found that using an entire possession resulted in lower prediction accuracy probably due to additional intricacies of court positions. Therefore, we use five second possessions for the remainder of this study.

Although each player has an assigned role, players change positions with respect to each other, which effectively changes their role during regular play (Lucey et al. (2013)). Since our classification problem is dependent upon a player's position, a player's role must be chosen

for each five second play. We ultimately decide to assign a player the role that they occupy at the beginning of the play. Since we want to explore how a player's actions resulted in favorable/unfavorable shooting position, we do not assign role based upon the end of the five second play.

2.5. Image-Based Representation of Player Trajectories

In terms of applying deep neural networks to multiagent trajectories, we first form an initial representation. A natural choice for representing these trajectories is in the form of an image. Given that the basketball court is 50x94 feet, we can form a 50x94 pixel image. In terms of the type of image we use, there are multiple choices: i) grayscale (where we identify that a player was a specific location by making that pixel location 1), ii) RGB (we can represent the offense trajectories in the red channel, the defense in the blue channel and the ball in the green channel, and the occurrence of a player/ball at that pixel location can be representing by a 1), and iii) 11-channel image (where each agent has their own separate channel) with each position represented by a 1. Examples of the grayscale and RGB approach are shown in Figure 2.1.

The 11-channel approach requires some type of alignment. In this paper, we apply the 'role-representation' which was first deployed by Lucey et al. (2013). The intuition behind this approach is that for each trajectory, the role of that player is known (i.e., point-guard, shooting guard, center, power-forward, small-forward). This is found by aligning to a predefined template which is learnt in the training phase.

Figure 2.1 shows examples of the methods we use to represent our data for our CNN. The grayscale image, which appears on the left, can accurately depict the various trajectories in our system. However, because the image is grayscale, the CNN will treat each trajectory the same.

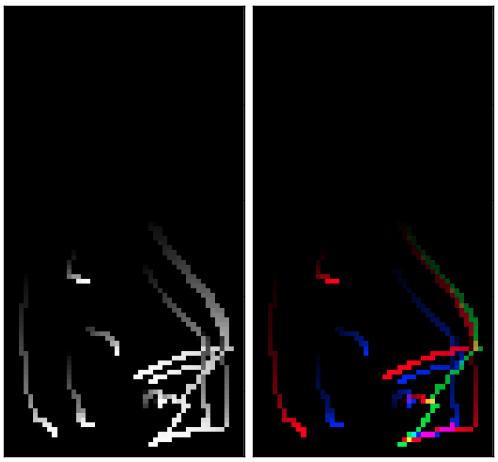

Figure 2.1. A grayscale and RGB image of the same trajectory of all 10 players plus the ball. Each red/blue line corresponds to an offensive/defensive player, and green indicates the ball trajectory.

Since we have an adversarial multiagent system in which defensive and offensive behavior in trajectory data can lead to different conclusions, grayscale is not the best option for representation. Therefore, to increase the distinction between our agents, we represent the trajectories with an RGB scale. We choose red to be offense, blue to be defense, and green to be the ball. This approach takes advantage of multiple channels to allow the model to better distinguish our adversarial agents. Although the ball may be part of the offensive agent structure, we decide to place the ball in a channel by itself since the ball is the most important agent. This approach, although better than the gray images, lacks in distinguishing player roles. Since we classify our made and missed shots along with the role of the player that shoots the ball, a CNN will have trouble distinguishing the different roles on the court. Therefore, for our final representation, we decide to separate all agents into their own channel so that each role is properly distinguished by their own channel.

The above ideas nearly create ideal images; however, it does not include time during a play. Since each trajectory is of equal brightness from beginning to end, it may be difficult to

identify where the ball was shot from and player locations at the end of the play. Therefore, we implement a fading variable at each time frame. We subtract a parameterized amount from each channel of the image to create a faded image as seen in Figures 2.1 and 2.2. Thus, it becomes trivial to distinguish the end of the possession from the beginning and leads to better model performance.

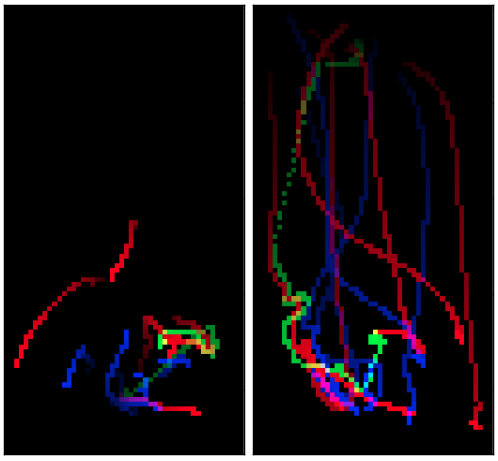

Figure 2.2. Figure on left depicts a five second play with all 10 players while the right is of an entire possession. As expected of five second plays, most of the trajectories remain near the basket located at the bottom of the image. Each red/blue line corresponds to a offensive/defensive player with green indicating the ball trajectory.

2.6. Prediction Models for Shot Prediction

To fully utilize the power of this dataset, we implement a variety of networks for our prediction task. For our base model, we use logistic regression with 197 hand-crafted features detailed later. To improve upon this basic model, we use a multi-layer FFN with the same features and utilize batch normalization during training. Because of the nature of these two models, we could only include the positions of the players at the time of the shot. Therefore, we craft images to include the position of the players throughout the possession. We then apply a CNN to

these new image features. Finally, we create a combined model that adopts both images and the original FFN features for training.

2.6.1. Logistic Regression and Feed Forward Network

For the baseline models, features based upon basketball knowledge were crafted. The list of features includes:

- Player and ball positions at the time of the shot
- Game time and quarter time left on the clock
- Player speeds over five seconds
- Speed of the ball
- Distances and angles (with respect to the hoop) between players
- Number of defenders in front of the shooter (30^0 angle of the shooter) and within six feet based upon the angles calculated between players
- Ball possession time for each offensive player
- Number of all individuals near the shooter (including teammates)

Logistic regression and FFN both use the same calculated features. In addition, only the CNN does not incorporate the above features.

A deep neural network is a machine learning model that consists of several layers of linear combinations of weights, inputs, and activation functions. The number of weights θ, layers L, and activation functions a are specified before training with data X. The model outputs probabilities f and the error is calculated generally with the Kullback-Leibler divergence, a.k.a log loss function $KL(y||f)$, against the true values y. A deep neural network generally refers to a neural network that is at least three layers deep. The depth (number of layers) of network

versus the breadth (number of neurons) allows it to learn more complex features of the dataset. The following is a mathematical model of a deep neural network:

$$\min_{\theta} E_{(X,Y)} KL(Y||f(X,;\theta)) = E_{(X,Y)} \sum_i y_i \log f_i(X;\theta) \quad \text{Optimization Function}$$

$$z^0 = X \quad \text{inputs of the model}$$

$$z_i^{\ell}(X;\theta) = a^{\ell}\Big(\sum_{k=1}^{K} z_k^{\ell-1}\theta_{ik}^{\ell-1}\Big) \quad \text{for neuron } i \text{ in layer } \ell \text{ and } K \text{ neurons at layer } \ell-1$$

$$f_i(X;\theta) = \frac{e^{z_i^L \theta_i^{L^T}}}{\sum_m e^{z_m^L \theta_i^{L^T}}} \quad \text{softmax function for probability of each class } i$$

2.6.2. Convolutional Neural Network

A CNN is similar to a Feed Forward Network, except that instead of learning individual weights per neuron, the model learns many filters (which consist of weights) that are convolved with the incoming data and reduced in size by a pooling layer. Like FFN's, they consist of multiple layers. For our model, we use a CNN that consists of three full convolutional layers, each with 32 3x3 filters and a max-pooling layer with a pool-size of 2x2 following each convolutional layer. After the final pooling layer, there is a fully connected layer with 400 neurons. The network ends with an output layer consisting of a softmax function and ten neurons. In addition, we use the ReLU function for our nonlinearity at each convolutional layer and the fully connected layer. We also implement AlexNet, Network-in-Network, and Residual Networks, but we did not garner significant improvement from any of these models.

2.6.3. CNN + FFN Network

The final network implemented is a combination of both the feed forward and convolutional networks. For this model, we use both the feed forward features and the fading trajectory images from the CNN. The idea behind the combined network is to have the model identify trajectory patterns in the images along with statistics that are known to be important for a typical basketball game.

The CNN and FFN parts of the combined network have the exact same architecture as the stand-alone versions of each model. The final layers just before the softmax layer of each stand-alone network are then fully-connected to a feed-forward layer that consists of 1,000 neurons. This layer is then fed into the final softmax layer to give predictions. After performing experiments and measuring log loss, we found that adding layers to this final network or adding additional neurons to this layer did not improve our final results.

2.7. Shot Prediction Results of Trajectory Data

All the models (FFN, CNN, and CNN+FFN) use the typical log loss function as the cost function with a softmax function at the output layer. The weights are initialized with a general rule of $\pm\sqrt{1/n}$ where n is the number of incoming units into the layer. For training, we implement the batch stochastic gradient method utilizing batch normalization. Batch normalization is used on the convolutional and feed forward layers of the model. In addition, we train the models on a NVIDIA Titan X using Theano.

For the CNN and CNN+FFN networks we utilize our eleven channel images with fading. We randomly split our data into train, validation, and test sets. The training set contains 52,000

samples while the validation and test sets contain 10,000 samples each. All experiments are completed using the same data split.

To justify our image representation, we first evaluate our model with each of the following image sets: one (grey) channel, three (RGB) channels, and eleven channels (for each player and the ball). These sets are all assessed on the previously mentioned CNN architecture consisting of three convolutional layers. Figure 2.3 displays the log loss and error rate on both validation and test sets for each of our representations. The log loss and error rate show a dramatic difference in accuracy based upon each image representation. By both accuracy and loss metrics, using eleven channels is the best representation. The eleven channel representation minimizes the overlapping trajectory issue that hinders the one and three channel methods. Thus, with eleven channels, the CNN can more easily capture more relevant on-the-court features such as ball possession and passes.

In addition to the image representations of Figure 2.2, we made each trajectory a different color in RGB space, varying the strength of the fading effect, and including extra channels of heat maps depicting the final ball position (none of which outperformed the eleven channel method). Successfully representing trajectory data in sport that outperforms traditional metrics is a nontrivial problem, but is not further explored in this study.

Next, we evaluate both the accuracy and loss values for our FFN, CNN, and combined CNN+FFN models. A quick observation of the metrics represented in Figure 2.4 shows that the final combined model is the best predictor. While the performance of the eleven channel images is an improvement over our other proposed image representations, there remains potential progress since our FFN has only slightly lower classification accuracy on the test set.

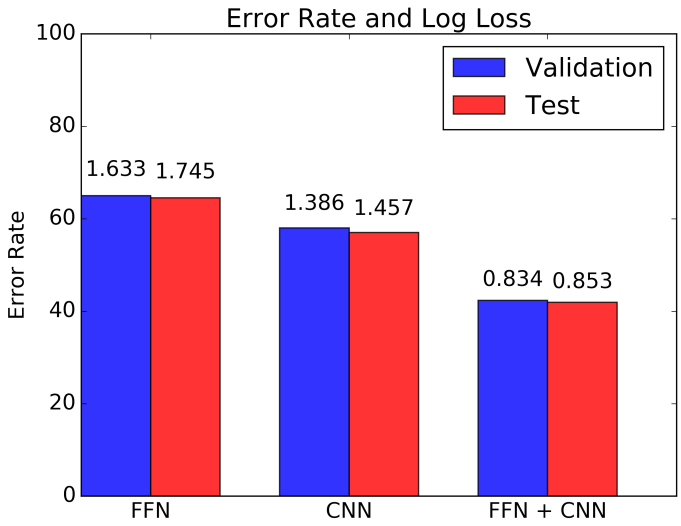

Figure 2.3. Log loss and accuracy for three image representations (smaller is better). The eleven channel method is superior.

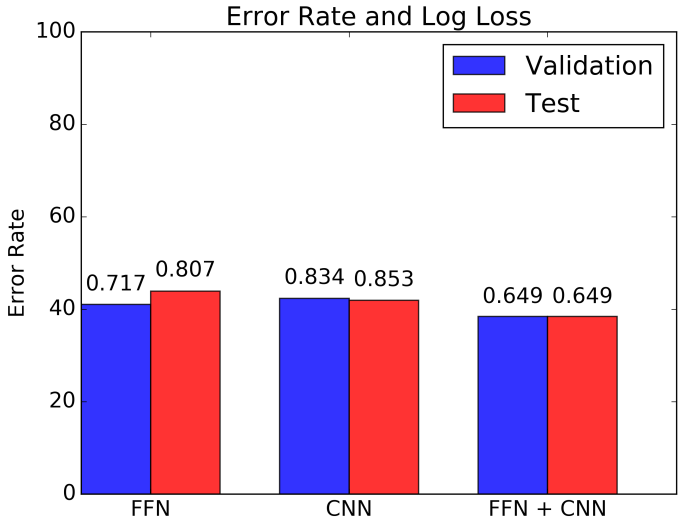

Figure 2.4. Log loss and accuracy for FFN, CNN, and combined CNN+FFN. The combined model is the best classifier by both metrics.

The remaining analyses are based on the combined CNN+FFN model. In addition to assessing the accuracy of our model, we explore a basic heat map of basketball shots based upon the raw data. At the very least, we expect that our complete model should be similar to a heat

map created via raw data. We make the heat map by taking a count of shots made against shots missed within a square foot of the basketball court. Since our classification model gives probabilities of making a shot (rather than a binary variable), we take the maximum probability to create a heat map equivalent to the raw data map. From the heat map comparison, we aim to further understand how our model is making predictions.

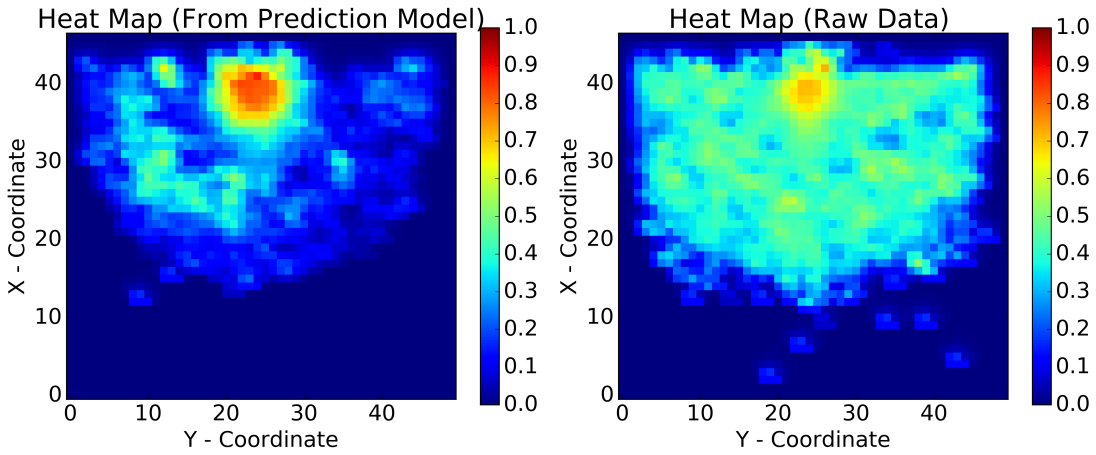

Figure 2.5. Heat map from model data and heat map of the raw data. The hot areas depict locations in which players are much more likely to take a shot.

In the raw data heat map, Figure 2.5, we note that the best probability of making a shot lies on top of the basket. As we get farther back, the probability decreases with two less probable zones (lighter zones separated by a thin green strip of higher probability): one right outside the paint and another just inside the three-point line. This means that our model is placing too much importance on the shooter being near the hoop. The model also predicts a larger high value area surrounding the basket, which extends further into the paint of the court. However, the dead zones in the model heat map are much larger. In addition, the model over predicts near the basket while under predicting outside this area. Curiously, there are a few areas that have higher probability outside of the paint.

To further explore our results, we create heat maps solely based on the role of the player (to break down scoring chances by agent). As before, each role represents an offensive player. In Figure 2.6 we present a few player roles and their representative heat maps. Role 3 must be the center position from their shot selection and Role 5 is the left guard. Note that the model predicts a much smaller area of midrange scoring probability than from the raw data for Role 3. The model heat map for Role 3 strictly covers the paint, while the raw data has significantly higher shot probabilities outside of the paint. Roles 1, 4, and 5 show similar behavior in the model prediction. These maps are very heavy-handed with respect to under the basket shots with extremely small probabilities outside of the paint. The one exception is Role 2, which the model predicts has a much more likely chance of scoring outside of the paint. We observe from these heat maps that Role 2 is the reason the heat map for all in Figure 2.5 exhibits an arc of higher probability outside the hoop. Therefore, for Role 2, our model is learning more than simply distance from the hoop for prediction in this case. The raw data shows that shots outside the hoop are just as likely from any player, but our model argues that Role 2 is more likely to make a shot when shooting further away from the hoop.

The probabilities that the combined model predicts for each shot may also provide useful insight into the game and our model's interpretation of high versus low value shots. We create these histograms by finding which examples the model gives the highest probability as a shot made or missed by player with role x. We then group these examples together, and the probability of making a shot is reported in the histogram. In the histograms depicted in Figure 2.7 we see that most of shots have a low probability. This agrees with common basketball knowledge because many of a guard's shots are beyond the paint. On the other hand, a center remains primarily under the basket and in the paint. Therefore, many of their shots are much more likely.

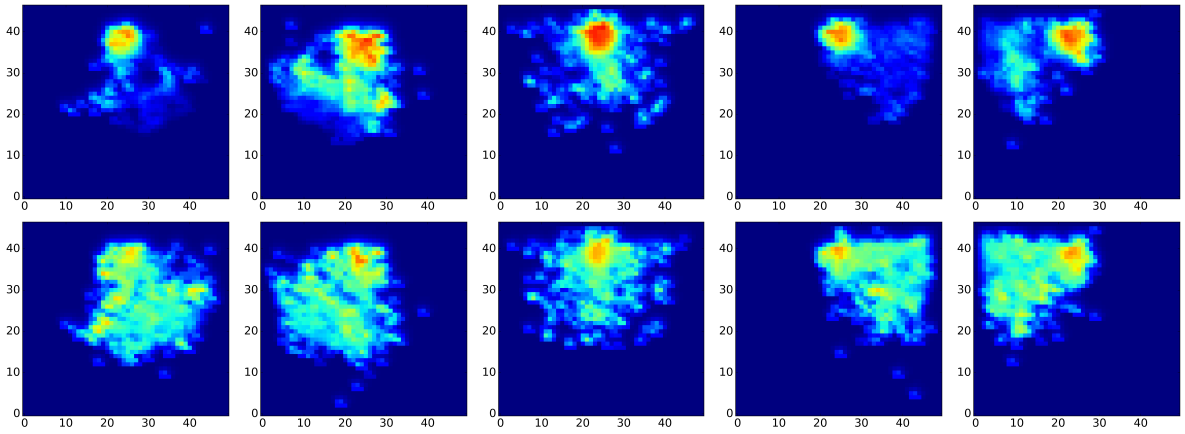

Figure 2.6. Roles 1-5 left to right. Model results top and raw data bottom. The model matches best with the center (image column 3) and power forward (image column 2), but provides a larger coverage in all other positions.

Watching a game live, a center getting a clean pass right under the basket often results in a made shot. In addition, most roles tend to follow the probability pattern of Role 1 except for Role 3 (the center), which has a wider distribution and higher average probability of making a shot. In addition, Role 5 tends to have a better ratio of high probability shots to low compared to Roles 1,2, and 4. A brief glance at NBA statistics agrees with this interpretation as the players with the highest shooting percentage (barring free throws) tend to be centers.

When viewing the histograms more carefully, there are additional role dependent insights. For example, Role 1 has the lowest probabilities for shots made compared to all other roles. Although the general shape is the same as for Role 2, the predictions are shifted significantly to the left. We also note that overall, most of the shots fall into the 45% category of shot being made, which is aligned with general basketball knowledge. Since these histograms are predictions of the average basketball player in each role, the takeaway message is that unless you have an ace shooter (Lebron James or Steph Curry), a team should focus on ball movement to give the center an open shot rather than relying on outside shooters.

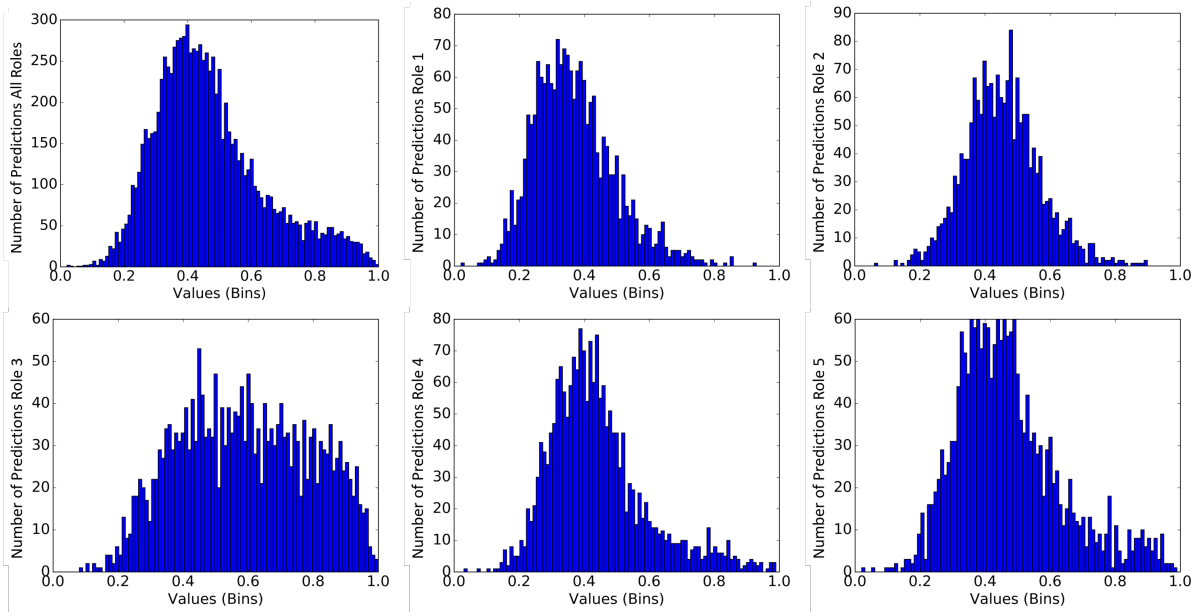

Figure 2.7. From left to right and top to bottom: (1) All roles. (2) Role 1. (3) Role 2. (4) Role 3. (5) Role 4. (6) Role 5. Probabilities of shot prediction by role. Note that role 3 (the center) has the largest overall probability of making a shot.

We also exhibit Figure 2.8 to provide additional visual context for our model probabilities. These figures depict the final positions of all players on the court at the time of the shot. We can assess how "open" the shot maker is at the time of the shot and the relative position of both the offensive and defensive players. The offensive players are blue, defensive players red, and the ball is green. Each offensive and defensive player has the letter "O" and "D" respectively followed by a number signifying the role of that player at that time. There are times when the model makes some questionable predictions. For example, the three-point shot that is exhibited in the top right of Figure 2.8 with a 0.610 probability is much too high. Unguarded, we do not expect three-point shots to be made more than 50% of the time. Thankfully, these examples are very rare in our model. For the most part, three-point shots are rated extremely low by the model garnering probabilities of less than 20%.

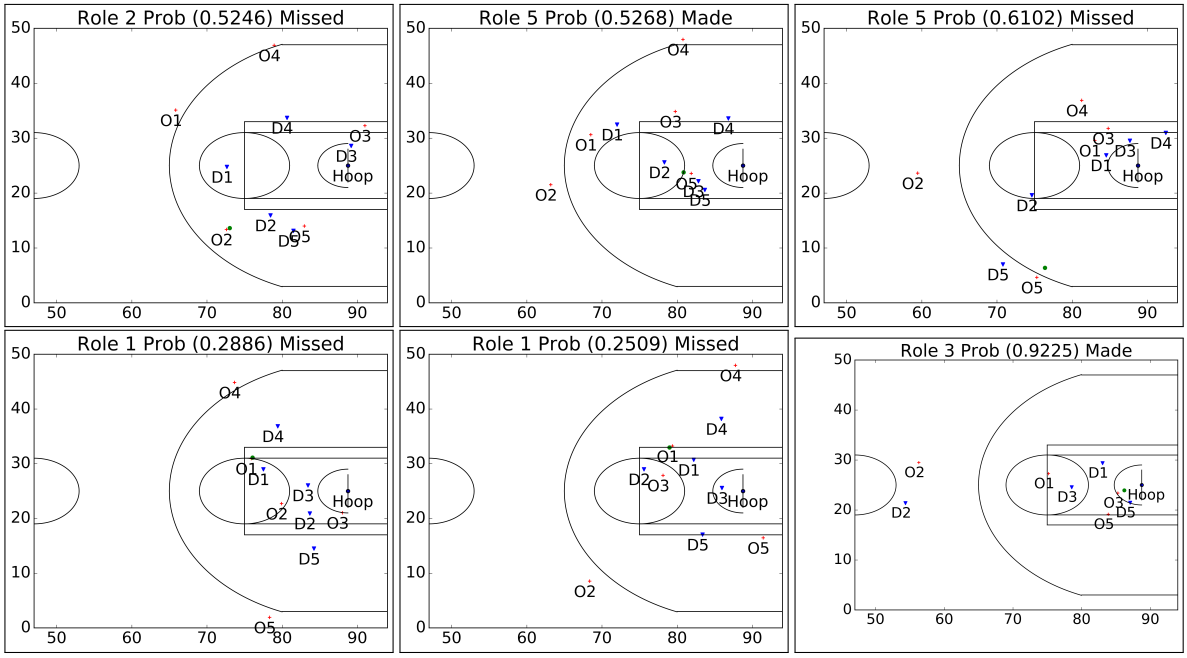

Figure 2.8. Locations of offensive (O) and defensive (D) players on the court at the time of the shot displaying the model's prediction probabilities. Most of the predictions are reasonable when considering shot location and defensive positioning.

In addition to three-point shots having a generally low probability, shots that are well-covered by defenders have a much lower probability of success. This is an unsurprising well-known result, but it does add validity to our model. In addition, shots that are open and close to the basket are heavily favored in our model. For example, in the bottom right picture, Role 3 has a very good chance of making a wide-open shot with the defense well out of position.

As noted in Figure 2.4, the FFN classifies with nearly the same accuracy as the CNN; however, the combined CNN+FFN model is the best classifier. Therefore, the CNN must learn features that elude the FFN.

We next explore our CNN model by creating images that result in maximum activation in the CNN (Erhan, Bengio, Courville, and Vincent (2009)). The goal of creating these images is

to find the features in our images that the CNN model learns. The process is similar to a reverse of training a neural network. First, we take an already trained CNN and a randomly created image. Then without changing the weights θ^* of the CNN that feed into filter i at layer ℓ, we use gradient ascent with respect to the image x that yield maximum value to activation a_i^ℓ. To make the image from the filter we solve: $x^* = \arg\max_x a_i^\ell(\theta^*, x)$.

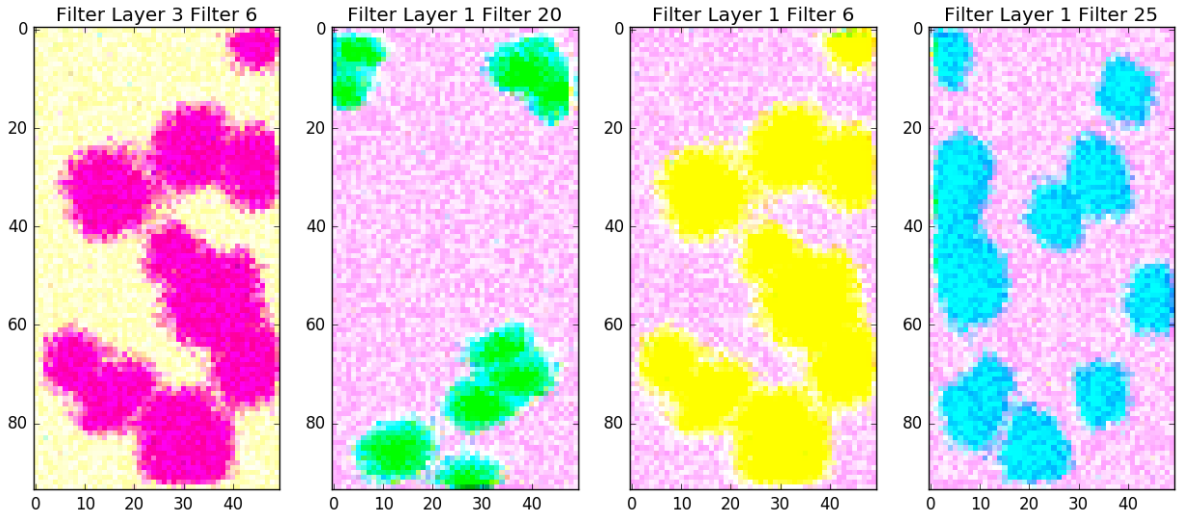

Figure 2.9. From left to right: (1) Filter 6 in convolution layer 3 of the offensive players (2) Filter 20 in convolution layer 1 of the ball (3) Filter 6 in convolution layer 1 of the offense and ball (4) Filter 25 in convolution layer 1 of the defense. Images that yield maximum activation using a trained CNN.

In Figure 2.9, we present four images from four distinct filters in our CNN model (several other filters result in white noise). The images in the figure are an RGB representation of the eleven channel images we create with the maximum activation method. Since we know which agent is represented by each channel, we let green, red, and blue represent the ball, offense, and defense, respectively. We implement this transformation into the RGB space for qualitative and quantitative assessments when compared to historical data of the ball, offense, and defense.

The first and third images of Figure 2.9 are nearly identical; however, while the first image displays primarily offensive areas, the third image presents the same areas but with ball (green) information as well. The second image shows that the filter is attempting to identify ball (green) activity, and the fourth image is a filter identifying defensive (blue) activity. We did not expect filters to look for offensive, defensive, or ball activity near the top of the court (away from the hoop) since we ensure that the offense always shoots towards the bottom of the image.

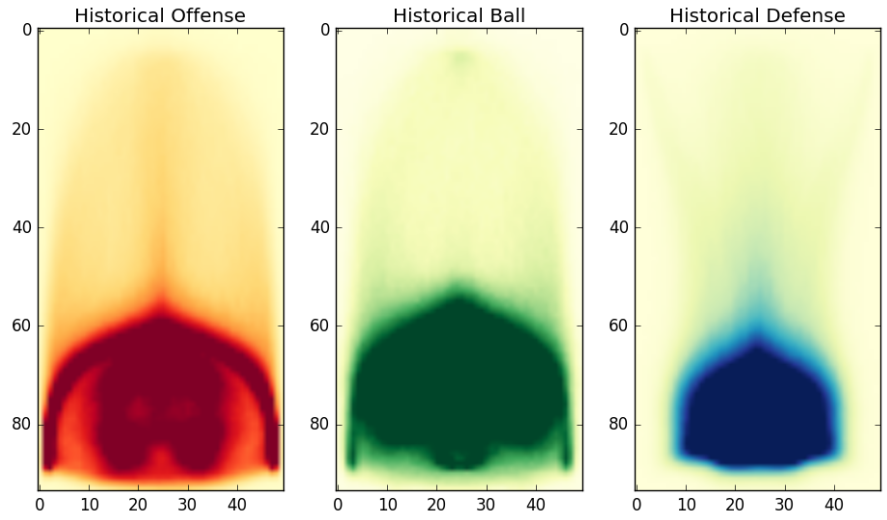

Figure 2.10. From left to right: (1) Historical offense data (2) Historical ball data (3) Historical defense data. Figures on the court of hot spots for the offense, ball, and defense at the time of a shot. Note that the defense is much more tightly packed around the hoop than the offense.

Figure 2.10 is a historical representation of the locations of all offensive (including the ball) and defensive agent locations at the time of the shot for our five second plays. For ease of comparison, we retain our qualitative RGB representation of green, red, and blue representing the ball, offense, and defense, respectively. The historical images in Figure 2.10 show all activity is near the hoop while the filter images in Figure 2.9 show additional activity far away from the hoop. To compare Figures 2.9 and 2.10 quantitatively, we utilize the SSIM, a.k.a. the structural

similarity index measure (Wang, Bovik, Sheikh, and Simoncelli (2004)). If a noisy version of an image is compared to the original, the SSIM can correctly identify that the images are the same. SSIM ranges from -1 to 1, where a perfect score of 1 indicates that the two images are the same.

We choose to compare the images from Figures 2.9 and 2.10 based upon our RGB representations. For example, previously we established that the red areas in image (1) of Figure 2.9 are a representation of the offense. Therefore, we compare the offensive channel of image (1) in Figure 2.9 to the historical offensive data of image (1) in Figure 2.10. We then compare image (2) of Figure 2.9 to image (2) of Figure 2.10 (historical ball data). We choose to compare image (3) of Figure 2.9, which we stated before represents both offensive and ball activity, to both the historical ball (2) and offense images (1) of Figure 2.10. After computing the SSIM for each part of the third filter of Figure 2.9, we average the two SSIM scores. Last, we compare image (4) of Figure 2.9 to historical defensive data displayed in image (3) of Figure 2.10. We calculate the SSIM for two cases: the entire court and the half of the court containing the hoop. We choose these two cases because the filter images of Figure 2.9 show activity away from the court while the images of Figure 2.10 do not. We detail the comparison images and resulting SSIM scores in Table 2.2.

Table 2.2. SSIM Results

Figure 2.9 Image	Figure 2.10 Image	SSIM (Half)	SSIM (Full)
Filter (1)	Offense (1)	0.623	0.715
Filter (2)	Ball (2)	0.310	0.277
Filter (3)	Offense, Ball (1,2)	0.554	0.580
Filter (4)	Defense (3)	0.561	0.680

When calculating the SSIM for both half court and full court images, we expected that the full court image scores would be lower. However, as seen in Table 2.2, this is not the case for a majority of the images. Historical data from Figure 2.10 shows that most activity is near the hoop; however, there is some activity far away from the hoop (likely due to transition plays). Since the filters of Figure 2.9 capture this, the full court SSIM is larger for three out of four of the Figure 2.9 images.

The worst performing image using SSIM as our evaluation tool is image (2) from Figure 2.9 with an SSIM of 0.310. This is due to a lack of large green areas near the hoop of the court in the filter image. When considering the entire court, image (2) from Figure 2.9 shows ball activity in the corners away from the hoop. Since the historical ball image in Figure 2.10 shows no activity in the upper corners, it is not surprising that it is the worst performing image.

Examining the results of Table 2.2, we can scrutinize the strength of our CNN. The accuracy of the CNN is close to that of the FFN, but the SSIM scores show room for improvement. It is clear that the CNN struggles to identify ball information when comparing image (2) of Figure 2.9 to historical ball data.

2.8. Discussion

Rather than other methods that incorporate transitions between a coarse and fine-grained approach or use small pieces of trajectory data, we utilize the full trajectory data of both offensive and defensive players. Since player alignments commonly permute, an image-based approach maintains the general spatial alignment of the players on the court. To integrate the time dependency of the trajectories, we introduce “fading” to our images to capture player paths. We found that using linear fading rather than a one-hot fade, works much better in our predictions.

In addition, using a different color for each trajectory does not increase predictability in an RGB, three channel, image. Since traditional CNN's utilize three channels, we found that networks created to work with that kind of data, such as residual networks, AlexNet, and Network in Network underperformed. We therefore opt to build our own CNN to handle the trajectory images. Thus, by using our combined CNN and FFN, we can predict whether a shot is made with 61.5% accuracy with the CNN proving to be more accurate than a FFN with hand-crafted features.

We found that by using a CNN, we can further explore the data using gradient ascent to picture the various filters of our network. We found that as the network gets deeper, it tends to gather several features together. For example, in the first layer, the filters look for shot locations. As we delve deeper into the network, the filters also begin to look for defensive and offensive spatial positions to make a more accurate prediction. Also, the histograms agree with common knowledge that centers have the highest short percentage since their shots tend to be right beside the basket.

For further research, it would be very interesting to identify time dependency in basketball plays. In our image data, we subtract a flat amount at each equally spaced frame to cause the fading effect. However, this assumes that the data in time is linearly related. Since this is not necessarily true, designing a recurrent model to find this temporal dependency could be a very interesting problem. Instead of having a fading effect in the image data, we can design an LSTM that takes a moving window of player and ball trajectories.

One last aspect that was not considered during this study was the identities of teams and players. The focus of this research was to gather more insight on the average shooting plays of

teams in the NBA. However, teams in the NBA have drastically different strategies. For example, the Golden State Warriors tend to rely on a three-point strategy while bigger teams, such the Thunder, build their offensive strategy around being inside the paint. Thus, new knowledge on basketball could be gathered if models were applied to different teams and possibly identify some overall team strategies. Such a more fine-grained analysis would require much more data.

CHAPTER 3

Activation Ensembles for Deep Neural Networks

3.1. Contribution

Many activation functions have been proposed in the past, but selecting an adequate one requires trial and error. We propose a new methodology of designing activation functions within a neural network at each layer. We call this technique an "activation ensemble" because it allows the use of multiple activation functions at each layer. This is done by introducing additional variables, α, at each activation layer of a network to allow for multiple activation functions to be active at each neuron. By design, activations with larger α values at a neuron is equivalent to being "chosen" by the network. We implement the activation ensembles on a variety of datasets using an array of FFN's and CNN's. By using the activation ensemble, we achieve superior results compared to traditional techniques. In addition, because of the flexibility of this methodology, we more deeply explore activation functions and the features that they capture.

3.2. Introduction

Most of the recent advancements in the mathematics of neural networks come from four areas: network architecture Jaderberg, Simonyan, Zisserman et al. (2015) Gulcehre, Moczulski, Denil, and Bengio (2016a), optimization method (AdaDelta Zeiler (2012) and Batch Normalization Ioffe and Szegedy (2015)), activations functions, and objective functions (such as Mollifying Networks Gulcehre, Moczulski, Visin, and Bengio (2016b)). Highway Networks Srivastava, Greff, and Schmidhuber (2015) and Residual Networks He et al. (2016) both use the

approach of adding data from a previous layer to one further ahead for more effective learning. On the other hand, others use Memory Networks Weston, Chopra, and Bordes (2015) to more effectively remember past data and even answer questions about short paragraphs. These new architectures move the field of neural networks and machine learning forward, but one of the main driving forces that brought neural networks back into popularity is the rectifier linear unit (ReLU)Glorot, Bordes, and Bengio (2011) Nair and Hinton (2010). Because of trivial calculations in both forward and backward steps and its ability to more effectively help a network learn, ReLU's revolutionized the way neural networks are studied.

One technique that universally increases accuracy is ensembling multiple predictive models. When neural networks garnered more popularity in the 90's and early 2000's, researchers used ensembles to great effect Zhou, Wu, and Tang (2002). Additionally, other techniques may identify as an ensemble. For example, the nature of Dropout Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014) trains many different networks together by dropping nodes from the entirety of a network.

We focus on activation functions rather than expanding the general architectures of networks. Our work can be seen as a layer or neuron ensemble that can be applied to any variety of deep neural network via activation ensembles. We do not focus on creating yet another unique activation function, but rather ensemble a number of proven activation functions in a compelling way. Each activation function, from ReLU to hyperbolic tangent contain advantages in learning. We propose to use the best parts of each in a dynamic way decided by variables that configure contributions of each activation function. These variables put weights on each activation function under consideration and are optimized via backpropagation.

As data passes through a deep neural network, each layer transforms the data to better interpret and gather features. Therefore, the best possible function at the top of a network may not be optimal in the middle or bottom of a network. The advantage of our architecture is that rather than choosing activations at specified layers or over an entire network, one can give the network the option to choose the best possible activation function of each neuron at each layer.

Creating an ensemble of activation functions presents some challenges in how to best combine the activation functions to extract as much information from a given dataset as possible. In the most basic model ensembles, one can average the given probabilities of multiple models. This is only feasible because the range of values in each model are the same, which is not replicated in most activation functions. The difficulty lies in restricting or expanding the range of these functions without losing their inherent performance.

An activation ensemble consists of two important parts. The first is the main α parameter attached to each activation function for each neuron. This variable assigns a weight to each activation function considered, i.e., it designs a convex combination of activation functions. The second are a set of "offset" parameters, η and δ, which we use to dynamically offset normalization range for each function. Training of these new parameters occurs during typical model training and is done through backpropagation.

Our work contains two significant contributions. First, we implement novel activation functions as convex combinations of known functions with interesting properties based upon current knowledge of activation functions and learning. Second, we improve the learning of any network considered herein, including the well-established residual network in the Cifar-100 learning task.

3.3. Review of Work on Activation Functions and Ensembling

There is a plethora of work in building the best activation functions for neural networks. Before ReLU activations were commonly used, deep neural networks were nearly impossible to train since the neurons would get stuck in the upper and lower areas of sigmoid and hyperbolic tangent functions. Some work has focused on improving these activation functions, such as Gulcehre et al. (2016a), who introduce a stochastic variable to the sigmoid and hyperbolic tangent functions. Since sigmoid and hyperbolic tangent function both contain areas of high saturation for values in large magnitude, the stochastic variable can aid in pushing the activation functions out of high saturation areas. In our work, rather than introducing stochasticity, we introduce several activations at each neuron, from which the network can choose a combination. Thus, it can reap the benefits of the sigmoid and hyperbolic tangent function without being limited to these functions at each layer.

On the other hand, Clevert, Unterthiner, and Hochreiter (2016) create a smoother leaky ReLU function dubbed the exponential linear function in order to take advantage of including negative values. Trottier, Giguère, and Chaib-draa (2016) further generalize the exponential linear unit by Clevert. The new exponential linear unit is called the Parametric Exponential Linear Unit (PELU). The extra parameters increase the flexibility of this activation function similar to that of the leaky ReLU compared to a regular rectifier unit. They find that this more general format dramatically increases accuracy. Our work differs from those who utilize different activation functions by being able to use as many activation functions as a user's computational capabilities allow. Rather than choosing one for a network, we allow the network to not only choose the best function for each layer, but also for each single neuron in each of the layers. We

grant activation functions with negative values in our network, but we restrict this after a simple transformation to ensure that our functions are similar in magnitude.

Li, Ouyang, and Wang (2016) take a different approach from typical work that focuses on activation functions. Rather than combining inputs, they use multiple biases to find features hidden within the magnitudes of activation functions. In this way, they can threshold various outputs to find hidden features and help filter out noise from the data. We restrict the range of our activation functions, which helps the neural network find features that may be hidden within the magnitude of another activation function. Thus, we are able to find hidden features via known activation functions without introducing multiple biases.

Scardapane, Scarpiniti, Comminiello, and Uncini (2016) create an activation function during the training phase of the model. However, they use a cubic spline interpolation rather than using the basis of the rectifier unit. Their work differs from ours in that we use the many different available activation functions rather than creating an entirely new function via interpolation. It is important to note that we restrict our activation function to a particular set and then allow the network to chose the best one or some combination of those available rather than have activation functions with open parameters (though this is possible in our architecture).

Maxout Networks, Goodfellow, Warde-Farley, Mirza, Courville, and Bengio (2013), allow the network to choose the best activation function similar to our work. However, Maxout Networks require many more weights to train. For each activation function, there is an entirely new weight matrix while our extra activations only require a few parameters. In addition, Maxout Networks find the maximum value for the activation function rather than combining the activation functions into a novel function as done in our work. Thus, features in a Maxout Network are lost due to not being represented after the maximization function.

Jin, Xu, Feng, Wei, Xiong, and Yan (2016) approach the problem of activation functions through the lens of ReLU's. Similar to PReLU's, which are leaky ReLU's with varying slope on the negative end, they create unique activation functions for rectifier units. However, they combine the training of the network and learning of the activation functions similar to our work and that of Scardapane et al. (2016). Their work restricts the combination to a set of linear functions with open parameters, while our does not have a restriction on the type of functions. Since the work of Jin et al. (2016) has this restriction, they cannot combine functions of various magnitude like the work we present here.

Chen (2016) uses multiple activation functions in a neural network for each neuron in the field of stochastic control. Similar to our work, he combines functions such as the ReLU, hyperbolic tangent, and sigmoid. To train the network, Chen uses Neuroevolution of Augmenting Topologies (NEAT) to train his neural network for control purposes. However, he simply adds together the activation functions without capturing magnitude and does not allow the network to choose an optimal set of activations for each neuron.

Work by Agostinelli, Hoffman, Sadowski, and Baldi (2015), which is closest to our approach, explores the construction of activation functions during the training phase of a neural network. Their work combines a rectifier unit with trainable variables to create a new function composed of rectifier units. Thus, rather than combining various activations functions together via an optimization process, Agostinelli et al. (2015) take the baseline foundation of the ReLU function and optimize based upon this singular function. The difference in our work is that we want to capture the features extracted by many different common activation functions to find the best possible activation function/s by neuron.

3.4. Activation Ensemble Architecture

Ensemble Layers were created with the idea of allowing a network to choose its own activation for each neuron and for each layer of the network. Overall, the network takes the output of a previous layer, for example from a convolutional step, applies its various activations, normalizes these activations, and places weights on each activation function. We first go through each step of the process of making such a layer.

The first naive approach is to simply take the input, use a variety of activation functions, and add these activation functions together. We denote the set of activation functions we use to train a network $f^j \in F$. Using this method, a network may reap the benefits of having more than one activation function, which may extract different features from the input. However, simply adding poses a problem for most functions. Many functions, like the sigmoid and hyperbolic tangent possess different values, but they can be easily scaled to have the same range. However, other functions, such as the rectifier family and the inverse absolute value function cannot be easily adjusted due to their unbounded ranges. If we simply add together the functions, the activation functions with the largest absolute value will dominate the network leaving the other functions with minimal input.

To solve this issue, we need to normalize the activation functions with respect to one another in order to have relatively equal contribution to learning. One option would be to use mean and standard deviation normalization; however, this would not equalize contribution. Therefore, we scale the functions to [0,1]. While building our method, we additionally performed tests using the range of [-1,1] for each activation function. We found that the performance was either close to that of [0,1] or slightly worse. In addition, allowing negative values causes additional issues when choosing the best activation functions with the α parameter we introduce later.

Simply adding activations together and forcing them to have equal contribution does not solve our second goal of finding the best possible activation functions for particular problems, networks, and layers. Therefore, we apply an additional weight value, α, to each activation for each neuron. Therefore, for the output of each neuron i and m being the number of activation functions, we have the following activation function for each neuron:

$$h_i^j(z) = \frac{f^j(z) - \min_k(f^j(z_{ki}))}{\max_k f^j(z_{ki}) - \min_k f^j(z_{ki}) + \varepsilon}$$
$$y_i(z) = \sum_{j=1}^{m} \alpha_i^j h_i^j(z)$$

Here, ε is a small number, k goes through all training samples (in practice k varies over the samples in a minibatch), and z_{ki} is the input to neuron i with training example k. We consider α_i^j as part of our network optimization. In order to again avoid one activation growing much larger than the others we must also limit the magnitude of α. Since the activation functions are contained to [0,1], the most obvious choice is to limit the values of α to lie within the same range. Thus, the network also has the ability to choose a particular activation function $f^j \in F$ if the performance of one far outweighs the others. For example, during our experiments, there were many neurons that we found heavily favored the ReLU function after training signified by the α^j for the ReLU function being magnitudes larger than the others.

In order to force the network to choose amongst the presented activation functions, we further require that all the weights for each neuron add to one. This then gives us another optimization problem to solve when updating the weights. In what follows, the values $\hat{\alpha}^j$ are the proposed weight values for the activation function j gradient update and α^j are the new values we find after solving the projection problem. We omit the neuron index i for simplicity, i.e. this problem must be solved for each neuron.

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \sum_{j=1}^{m} \tfrac{1}{2}(\hat{\alpha^j} - \alpha^j)^2 \\ \text{subject to} \quad & \sum_{j=1}^{m} \alpha^j = 1 \\ & \alpha^j \geq 0,\ j = 1, 2, \ldots m. \end{aligned}$$

This optimization problem is convex, and is readily solvable via KKT conditions. It yields the following solution:

$$\alpha^j = \begin{cases} \hat{\alpha}^j + \lambda, & \text{if } \lambda > \alpha^j, \\ 0, & \text{otherwise.} \end{cases}$$

$$\sum_{j=1}^{m} \max(0, \hat{\alpha}^j + \lambda) = 1$$

This problem is very similar to the water-filling problem, and has an $O(m)$ solution. Algorithm 1 exhibits the procedure for solving this problem. The projection sub-problem must be solved for each neuron for each layer that our multiple activations are applied.

Algorithm 1 Projection Sub-Problem

Input: Vector $\hat{\alpha}$,
Initialize Vector $G = 1$ and Scalar $D = m$
for $k = 1$ **to** m **do**
  $\lambda = \frac{1 - \sum_{j=1}^{m} \hat{\alpha}^j}{D}$
  $\alpha = \hat{\alpha} + \lambda$
  $\alpha = \alpha \circ G$ (Hadamard Product)
  if any $\alpha^k < 0$ **then**
    $D = D - 1$
    $G_k = 0$
  end if
end for

We borrow two elements in our approach from Batch Normalization Ioffe and Szegedy (2015). We record running minimum and maximum values while training over each minibatch. Thus we transform the data using only a small portion (dictated by batch-size during training). The other element that we introduce is the two parameters, η and δ, which allow for the possibility of the network choosing to leave the activation at its original state. Below is the final equation that we use for the activation function at each neuron.

$$y_i = \sum_{j=1}^{m} \alpha^j (\eta^j h_i^j + \delta^j)$$

Therefore, the final algorithm for our network involves both maintaining running maximum and minimum values and solving the projection subproblem during training. Then in the test phase, we use the approximate minimum and maximum values we obtain during the training phase. The weights learned during the training phase, namely our new parameters α, η, and δ remain as is during training. In summary, rather than creating a new weight matrix for each activation function, which adds a huge amount of overhead, we allow the network to change its weights according to the activation function that is optimal for each neuron.

We next provide derivatives for the new parameters. Let ℓ denote the cost function for our neural network. We backpropagate through our loss function ℓ with the chain rule to find the following:

$$\begin{aligned}
\frac{\partial \ell}{\partial \alpha^j} &= \frac{\partial \ell}{\partial y_i} \cdot \left(\eta^j h_i^j + \delta^j \right) \\
\frac{\partial \ell}{\partial \eta^j} &= \frac{\partial \ell}{\partial y_i} \cdot \alpha^j h_i^j \\
\frac{\partial \ell}{\partial \delta^j} &= \frac{\partial \ell}{\partial y_i} \cdot \alpha^j
\end{aligned}$$

As a note, activation ensembles and the algorithms above work in the same way for CNN's. The only difference is that instead of using a set of activations for a specific neuron, we use a set of activations for a specific filter.

3.4.1. Activation Sets

To explore the strength of our ensemble method, we create three sets of activations to take advantage of the weakness of individual functions. The first set is a number of activation functions seen in networks today. One of the functions, the exponential linear units, garners favorable results with datasets such as CIFAR-100. Others include the less popular inverse absolute value function and the sigmoid function (which is primarily relegated to recurrent networks in most literature). One omission we must mention is the adaptive piecewise linear unit. Since our goal is to create an activation from common functions, a function that focuses on adapting via weights is not included. The following are the activations we use for the first set: sigmoid, hyperbolic tangent, soft linear rectifier, linear rectifier, inverse absolute value, and the exponential linear function.

The next two sets of activations are designed to take advantage of the ensemble method's ability to join similar functions to create a better function for each neuron. Since ReLU functions are widely regarded as the best performing activations functions for most datasets and network configurations, we introduce an ensemble of varying ReLU functions. Since rectifier neurons can "die" when the value drops below zero, our ensemble uses rectifiers with various intercepts of form $f^b(z) = \max(0, z+b)$, where $b \in \mathbb{R}$. We settle with five values for b, $[-1.0, -0.5, 0, 0.5, 1.0]$, to balance around the traditional rectifier unit.

The final activation set is a reformation of the absolute value function. It only consists of two mirrored ReLu functions of the form, $f^1(z) = \max(0, -z)$ and $f^2(z) = \max(0, z)$. The behavior of the graphed function is very similar to that of an absolute value; however, our function contains individual slopes that vary by the value of α_1 and α_2. We create this function to capture elements of the data that may not necessarily react positively with respect to a rectifier unit, but still carry important information for prediction.

3.5. Results of Activation Ensembles on Prominent Datasets

For our experiments, we use the datasets MNIST, ISOLET, CIFAR-100, and STL-10. For MNIST and ISOLET, we use small designed feed forward and convolutional neural networks. For CIFAR-100, we apply a residual neural network. Last, we design an auto-encoder strategy for STL-10. In addition, we use Theano and Titan X GPU's for all experiments. We found that the optimization function AdaDelta performs the best for all of the datasets. For each network, we set the learning rate for AdaDelta to 1.0, which is the suggested value by the authors for most cases. In each network that was designed in-house, we implemented batch normalization before each ensemble layer. We describe the architecture used for each dataset under their respective sections. The nomenclature we use for network descriptions is $(16)3c-2p-10f$ where $(16)3c$ describes a convolutional layer with 16 filters of size 3x3, $2p$ for a max-pooling layer with filter size 2x2, and $10f$ for 10 neurons of a feed foward layer.

The weights at the ensemble layer were initialized at $\frac{1}{m}$ where m is the number of activation functions at a neuron. Furthermore, we set $\eta = 0$ and $\delta = 1$ and initialized the traditional neural network weights with the Glorot method. In addition, we initialize the residual network using the He Normal method. We train on our three activation sets as well as the same networks

with rectifier units for the original networks since they are the standard in most cases. Our stopping criterion is based upon the validation error for each network except for the residual network, in which the suggested number of epochs is 82. Also, we apply AdaDelta for each optimization step for all new variables as well, α, η, and δ for each minibatch. In Table 3.1 below, we summarize the test accuracy (reconstruction loss for STL-10) of our datasets and various models. Each number is an average over five runs with different random seeds. Note that the largest improvement is found for the ISOLET dataset; however, we also find that separate models and datasets prefer one activation function over another, which we expand upon below.

Table 3.1. Final Model Results for Models with and without Activation Ensembles

DataSet	Model	Original/Ensemble
MNIST	FFN	97.73% / 98.37%
MNIST	CNN	99.34% / 99.40%
ISOLET	FFN	95.16% / 96.28%
CIFAR-100	Residual	73.64% / 74.20%
STL-10	CAE	0.03524 / 0.03502

3.5.1. Comparing Activation Function Sets

We first explore the α parameter values of our activation functions. We primarily concentrate on the first set of activations (Sigmoid, Tanh, ReLU, Soft ReLu, ExpLin, InvAbs). Since ReLU is the most common activation function in literature, we expect it be chosen the most by our networks. As seen Figures 3.1,3.3, and 3.4, we find this to be true. However, neurons that are deeper may not choose any particular activation. In fact, at some neurons in the bottom layers, the parameters for choosing a function are nearly equal.

Figure 3.1 illustrates the differences between layers of our activation ensembles for the in-house Feed-Forward Network model applied to the MNIST dataset. Each data point is taken at the end of an epoch during training. The neurons in the images were chosen randomly.

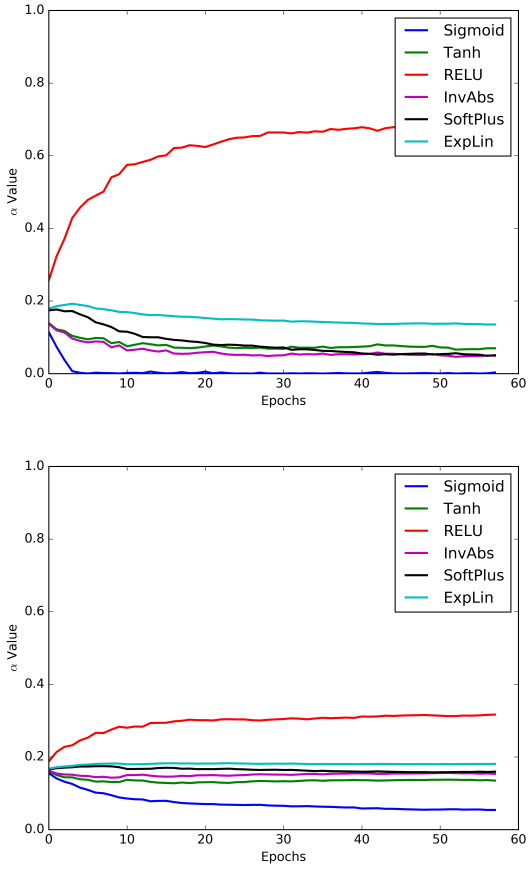

Figure 3.1. Top image is a neuron from the first layer while the bottom is from the second layer

The top image of Figure 3.1, which comes from the top layer of the network, shows that the ReLU becomes the leading function very quickly. The other functions, with the exception of the exponential linear function, are very close to zero. The bottom image, which is a neuron from the next layer in the network, still chooses the rectifier unit, but not as resolutely as in the

previous layer. It is interesting that the sigmoid function in the next layer does not become zero immediately like the neuron in the previous layer. We experimented as well with momentum, and found that with a step of 0.1, the network still chooses a leading activation function. We find that this is a general trend for FFN's and the MNIST dataset. However, as we discuss below, this does not hold for all models or datasets.

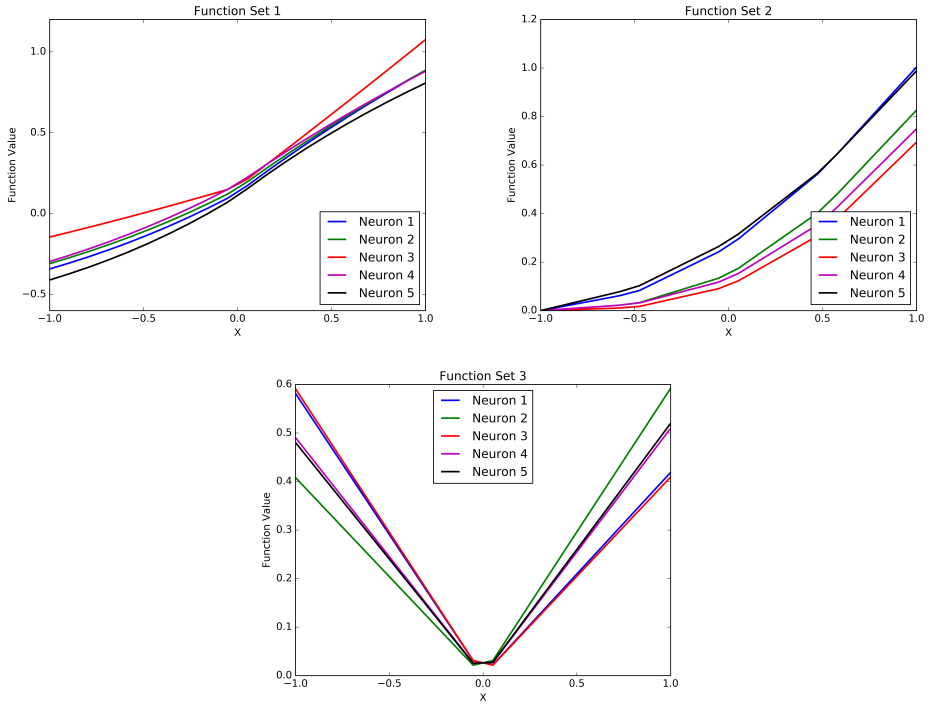

Figure 3.2. Example activation functions from each of the three sets of functions.

Next we compare the functions that the ensembles create. Figure 3.2 depicts the various sets of activations for our ensemble. The images are of the same five random neurons taken from the second layer of the FFN for the MNIST dataset in the first two figures while the final image is taken from the second layer of the same model trained on the ISOLET dataset. The images are made by taking the final α's for each activation function, inputing a uniform vector from $[-1,1]$,

and adding the functions together. Note that we leave out the normalization technique used for activation ensembles and the values for η and δ. The first set of activations, presented in the top left of Figure 3.2, behave very similarly to the exponential linear unit. However, this function appears closer to an x^3 function because of its inflection point near the origin. The second set of functions clearly form a piecewise ReLU unit with the various pieces clearly visible. The last set appears much like an absolute value function, but differentiates itself by having varying slopes on either side of the y-axis. We observe that the best activation function for each model is the third set, with the exception of the residual network.

3.5.2. MNIST

We solve the MNIST dataset using two networks, a FFN and a CNN. The FFN has the form $400f - 400f - 400f - 10f$ while the CNN is of the form $(32)3c - 2p - (32)3c - 2p - 400f - 10f$. MNIST is not a particularly difficult dataset to solve and is one of the few image-based problems that classical FFN's can solve with ease. Thus, this problem is of particular interest since we may compare results between two models that can easily solve the classification problem with our activation ensembles. One of the aspects we explore is whether or not different activations prefer certain models.

In Figure 3.3, we present histograms representing the share of α values for each activation function in the ensemble. The histograms are a representation of favored activation function by layer for the first set of activations. The top left image in Figure 3.3 shows what we expect, namely that the ReLU dominates the layer. As we move towards the bottom of the network, the ReLU is still the most favored function; however, the difference becomes marginal. Also note that the sigmoid function is the lowest value function in all 3 layers and that the exponential

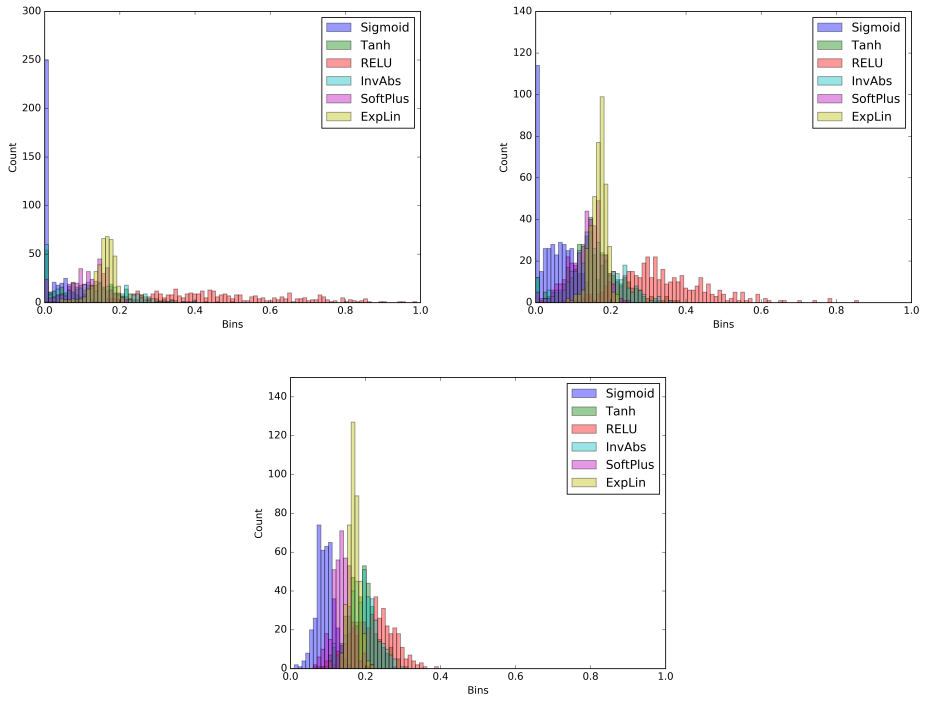

Figure 3.3. Left to right layers 1-2 and bottom layer 3 for a Feed Forward Network on MNIST of activation set 1.

linear function has the smallest range of all the functions. For the experiments on the other two activation sets, we note that the difference between one activation and another is not nearly as significant as activation set 1.

We also observe the α's in our CNN and found that the hyperbolic tangent and inverse absolute value functions overtake the rectifier unit in importance at the third layer (a feed-forward layer) versus the first two convolutional layers. Thus, it appears that the transition of one type of layer to another changes the favored activation functions. Therefore, we see that given a single dataset, the favored activation depends on the layer and model used for classification.

3.5.3. ISOLET

ISOLET is a simple letter classification problem. The data consists of audio data of people uttering letters. The goal is to predict the letter said for each example (a-z). ISOLET has 7797 examples and 617 attributes. For training and testing, we split the data into 70% for training and 30% for testing. In addition, we use our in-house FFN described in the previous section ($400f - 400f - 400f - 26f$). We train both a network with only ReLU's at each neuron and a network with activation ensembles. For ISOLET, we implement all three classes of ensembles we mention in the previous section.

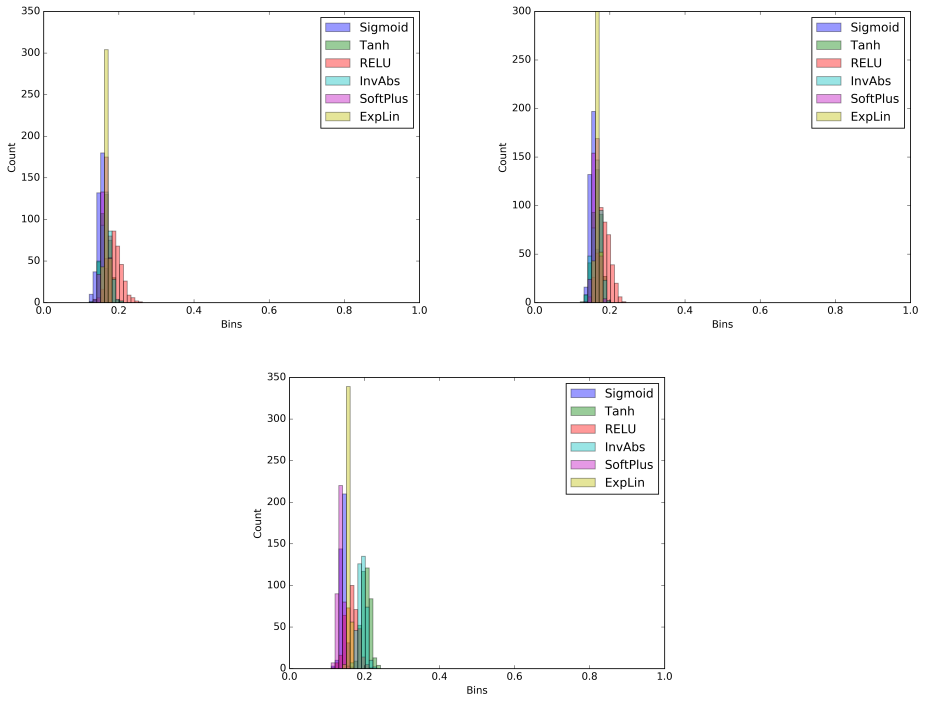

Figure 3.4. Left to right layers 1-2 and bottom layer 3 for a Feed Forward Network on ISOLET of activation set 1.

In Figure 3.4, we present the same images provided for the MNIST dataset. We see much different behavior of the α values than we did with the MNIST dataset. Most values are clumped

up near one another. Though the ReLU is the leading activation function, it is not by much in the first two layers. Similar to the CNN for MNIST, hyperbolic tangent and the inverse absolute value functions move up in importance.

Thus, we see that even with the same network and sets of activation functions, the model chooses different optimal activation functions based upon each individual dataset.

3.5.4. CIFAR-100

In order to get a broad scope of networks and test the strength of activation ensembles, we decide to use a residual network for CIFAR-100. The network we use comes from the Lasagne recipe Github <https://github.com/Lasagne/Recipes/tree/master/papers>. It was originally designed for the CIFAR-10 dataset, but we modify the model for CIFAR-100. The model has three residual columns of which include five residual block each followed by a global pooling layer for a total of 32 layers.

Challenges arise from two areas while implementing activation ensembles on residual networks. First, residual networks are specifically designed for ReLU's via the residual element and initialization. Second, the placement of each activation ensemble needs to be carefully constructed to avoid disrupting the flow of information within each residual block.

We attempt various approaches to activation ensembles on residual networks. The failed ideas include having an ensemble after each residual block and after the second stack but before adding in the residual. The implementation that works best is to incorporate the activation ensemble in the middle of a residual block and not at the end of the block or after. In addition, this is the only network in which the second class of activations (the 5 ReLU's with various

intercepts) works best. This is an expected result because of a residual network's dependence on the ReLU.

3.5.5. STL-10

The CAE (Convolutional Autoencoder) we implement for this problem is very similar to one done by Dosovitskiy, Springenberg, Riedmiller, and Brox (2014). However, our network contains less filters at each layer than their network due to insufficient GPU memory on our graphical cards. Our structure is the following for the encoder portion of the autoencoder: $(32)5c-2p-(64)5c-2p-(128)5c-6p$. Some CAE's use an ReLU at the end for prediction purposes, but we use the sigmoid function. We keep normalization very simple with this network by only scaling the data by 255 at the input level. The reproduced images by the network are then very easy for comparison by using the sigmoid function and mean-squared error. During preliminary experiments we saw certain classes of activation functions tend to perform better, and thus we implemented only the third ensemble set that resembles an absolute value function.

We additionally tie all the weights for the encoder and decoder to restrain the model from simply finding the identity function. Since tying the weights is suitable for autoencoders, we also tie the weights associated with our activation ensembles (this includes α, δ, and η values). We choose to not tie the maximum and minimum values we find during the normalization stage as those are simply used for the purpose of our projection algorithm for the α variables.

3.6. Discussion

In our work, we introduce a new concept we title as an activation ensemble. Similar to common ensembling techniques in general machine learning, an activation ensemble is a combination of activation functions at each neuron in a neural network. We describe the implementation for standard feed-forward networks, convolutional neural networks, residual networks, and convolutional autoencoders. We create a convex combination of activation functions yet to be seen in literature with an algorithm to solve the new projection problem associated with our model. In addition, we find that these ensembles improve classification accuracy for MNIST, ISOLET, CIFAR-100, and STL-10 (reconstruction loss).

To gain more insight into activation functions and their relationships with the neural network model implemented for a given dataset, we explore the α's for each activation function of each model. Further, we examine one dataset, MNIST, with two models, FFN and CNN. This enables us to discover that the optimal activation varies between a FFN and CNN on the same dataset (MNIST). While the top two layers of both favor the rectifer unit, the bottom layer prefers the hyperbolic tangent and inverse absolute value functions. When comparing the activation ensemble results between ISOLET and MNIST on the FFN, we observe that the optimal activation functions between datasets are also different. While MNIST esteems the ReLU, ISOLET does not favor a particular activation until the bottom layer, in which it chooses the hyperbolic tangent.

CHAPTER 4

Dynamic Time Prediction for Stock Prices

4.1. Contribution

Recurrent neural networks and sequence to sequence models require a predetermined length for prediction output length. Our model addresses this by allowing the network to predict a variable length output in inference. A new loss function with a tailored gradient computation is developed that trades off prediction accuracy and output length. The model utilizes a function to determine whether a particular output at a time should be evaluated or not given a predetermined threshold. We evaluate the model on the problem of predicting the prices of securities. We find that the model makes longer predictions for more stable securities and it naturally balances prediction accuracy and length.

4.2. Introduction

Recurrent neural networks are very popular and effective at solving difficult sequence problems such as language translation, creation of artificial music, and video prediction. New architectures, such as Sequence to Sequences networks by Sutskever, Vinyals, and Le (2014) and Memory Networks by Sukhbaatar, Weston, Fergus et al. (2015) are used to solve problems in language translation and answer questions using a large memory bank. However, these problems generally have training data with given sequence outputs (for example, a model translating a sentence from English to Spanish). Because input and output sequences are known a priori for these problems, it is possible to solve them with a fixed model architecture.

A fixed model architecture is effective for sequences, but there are a number of problems related to multiple time series datasets that do not have a natural sequence size. For example, a company may wish to predict the number of products to be shipped out for sale based upon customer demand. Each product has a different amount of demand volatility, which can make an enormous difference in how far in advance they are willing to predict the demand of a product. In this case, it would be extremely useful to have a model that can balance F1 score and the number of future predictions based upon a product's base demand.

Another example, which we explore in this work, is financial security price prediction. Some securities are extremely volatile, which makes prediction for longer times highly inaccurate. On the other hand, low volatility securities are easier to predict further into the future.

The biggest problem in multiple time series predictions when it comes to dynamic prediction length is that the training data does not exhibit output sequences of various length. For this reason, a different model is required. In multiple time series, input sequences can be naturally created for example by a fixed-size sliding window. However, the length of the output sequences can be dynamic since typically there is flexibility in how far to predict in the future. In inference, we allow our prediction model to generate a different number of predictions depending on the current input sequence as well as a different number of predictions per time series. The number of predictions the model generates depends on a thresholding function that determines the model's confidence of that particular output. If the confidence is too low, we no longer consider the predictions our model generates for that particular sample.

The main objective of our study is to create a prediction model that balances F1 score and predicting into the future. The main challenge is the fact that samples which are taken from infinite time series do not naturally contain dynamic length predictions. This aspect requires

a different loss function that includes the notion of confidence and tailored computation of gradients on different length output sequences. We create a model architecture relying on a novel loss function that allows for a dynamic number of output predictions. We explain the ideas and concepts by utilizing predictions of several correlated financial securities. In this case, rather than having to adjust the predictive length manually depending on market volatility, the model learns how far in advance it can confidently predict during training. For security prediction, a model that is not limited to a fixed number of output predictions can provide much more robust price predictions. For example, we expect that some security j with high volatility during the training phase should result in fewer predictions. On the other hand, if the same security j is trained during a low volatility period, we expect the model to generate more predictions. Clearly, a dynamic model that can easily adjust to the current training environment of a security can provide huge benefits. In inference the model provides a natural way to stop generating output predictions.

Our work contains two main contributions. First, we create a way to measure the confidence of a model's prediction without having to rely on Bayesian statistics. Second, our model is the first of its kind to allow for dynamic prediction length with sequence to sequence networks. Along the way, we have to tailor gradient computation.

In our study, we use two financial security datasets which consist of several years of tick prices. One contains five distinct securities and the other contains twenty-two different securities. We find that our new architecture successfully balances prediction F1 score and the number of future predictions. In addition, our architecture uses different prediction lengths at different times for each security due to stochastic drift between training and test sets. The best dynamic output prediction length model is a sequence to sequence network which earns an F1 score of

0.503 in contrast to a traditional LSTM structure that only gets an F1 score of 0.209 for a single prediction.

In Section 4.2, we review two main subjects related to our work. First, we inspect studies within the realm of deep learning related to our new model architecture. Second, we analyze other work on predicting financial securities with a focus on machine learning and deep learning methods. In Section 4.3, we present the dynamic prediction length model while in Section 4.4 we present a computational study based on securities.

4.3. Related Work

Similar to our concept of dynamic output prediction, Pointer Networks by Vinyals, Fortunato, and Jaitly (2015a) are used for problems such as sorting variable sized sequences. They use an attention mechanism that points to a particular part of the input sequence that is used as the next output. Although this architecture can allow for variable input sizes, the output size is constrained to be the same size as the input. Our model allows for any size output (unrelated to the input size) up to some arbitrary maximum size. Pointer networks are also not applicable to our case since output is not a specific part of input.

Graves (2016) introduces adaptive computation time (ACT) for recurrent networks. The author creates an additional metric that allows the network to continue "pondering" the input through additional computation. We can think of ACT as a model that in each time has a dynamic number of stacked LSTM or GRU cells. We considered using ACT along with our architecture, but it increases the computational complexity by recalculating the feed-forward step a number of times. Basing the stopping decision with respect to the natural choice of the output time requiring substantial computational time, does not work when most times a single

layer is needed. In time series with a walk forward strategy, smaller less complex models are more effective. This renders ACT not appropriate. In addition, online algorithms need to be agile, and adding computational time would be a detriment to a model. Therefore, we choose to use traditional LSTM architectures that train much faster.

To achieve better training and dynamic output sizes, we added an additional term to the loss function and modify the measure of predictions. Although there have been many new architectures such as Residual Networks by He et al. (2016), Memory Networks by Sukhbaatar et al. (2015), and Neural Turing Machines by Graves, Wayne, and Danihelka (2014), none of these incorporate a new loss function with their architecture. This tendency of not creating new loss functions is noted by Janocha, Czarnecki et al. (2017). The authors explain that although there is a lot of work in neural network activations, optimization, and architecture, the loss function used for nearly all neural networks is a combination of log loss and L1/L2 norms. Janocha et al. (2017) show that functions that were previously deemed to be inferior in deep learning can be more robust than log loss for classification problems. Therefore, it is important for studies to continue exploring loss functions to increase the network performance and to create a variety of new models.

There are some studies that significantly change the loss function in deep learning. For example, GAN's by Goodfellow, Pouget-Abadie, Mirza, Xu, Warde-Farley, Ozair, Courville, and Bengio (2014) are immensely popular and design a loss function for their specific problem and architecture to balance generation and classification. ACT by Graves (2016) also uses a unique addition to the loss function so that their network does not "ponder" on the input for too long. There is growing momentum for using the Wasserstein metric as the loss function as seen in the work by Frogner, Zhang, Mobahi, Araya, and Poggio (2015). The Wasserstein metric is

even successfully being used in GAN's in the work of Arjovsky, Chintala, and Bottou (2017). We expand the volume of work in this area by developing a loss function that encourages a model to have a dynamic output length at prediction. In contrast to loss functions specific to problem type, our architecture can be used with any recurrent neural network architecture.

Next, we focus on works on predicting financial securities with deep networks. Most work utilizing deep neural networks focuses on applying other forms of data for prediction, such as news about financial markets or specific companies. Chong, Han, and Park (2017) review many of the prediction methods commonly used for security prediction and the predicted outcome. Niaki and Hoseinzade (2013), focuses on predicting upward and downward movement of S&P 500 with a deep feed-forward network. Ding, Zhang, Liu, and Duan (2015) use historical pricing data in combination with financial news data with a deep feed forward network. Krauss, Do, and Huck (2017) utilizes both random forests and deep feed-forward networks to find statistical arbitrage on the S&P 500. In contrast to the aforementioned models, we predict significant movement in prices, utilize temporal models, and predict multiple securities with a single model.

Sirignano (2016) uses deep learning directly on financial security price data as well. He utilizes limit order book data with multiple ask/bid prices for each security to predict the change in the spread. In addition, he uses a deep feed-forward network and a separate model for the prediction of each security. In contrast, we predict all securities with one model and apply recurrent neural networks in addition to feed forward networks. Another similar work, Dixon, Klabjan, and Bang (2016), uses deep feed-forward networks for prediction directly on security prices. In contrast, our study applies recurrent and convolutional recurrent models in addition to the baseline feed forward network.

There is also work on security prediction beyond the standard feed-forward network. Borovykh, Bohte, and Oosterlee (2017) uses convolutional neural networks over the security time series to make predictions. Others, such as Bao, Yue, and Rao (2017), use stacked auto-encoders and wavelet transformations to form an embedding of financial data, and feed this into an RNN model for prediction. Chen, Zhou, and Dai (2015) use an RNN model on opening and closing security prices on the Chinese market with seven classification categories. Akita, Yoshihara, Matsubara, and Uehara (2016) use both textual and price information to predict a security price. In contrast to these works, our model consists only of security price data. Also, we apply sequence to sequence and convolutional LSTM models in addition to having a dynamic output length for security predictions.

4.4. Model

In this section, we explain the architecture of our model that predicts a dynamic number of outputs. The model uses a sequence to sequence (Seq2Seq) network in combination with our new proposed loss function. For details on Seq2Seq networks, please see the original paper by Sutskever et al. (2014). We also use attention in our model which is explained in the paper by Bahdanau, Cho, and Bengio (2014).

In addition, we use teacher forcing. We found that the first input to the decoder has a great impact on accuracy. By far the best performing first decoder input is the ground truth associated with the last encoder input.

For a general Seq2Seq network, let θ describe the trainable weights, X be our input sample, and $f_t^q(X, \theta, f_{i<t})$ be the final output via a softmax function at time t for time series q, $q = 1...Q$. In the presence of several time series, each time series requires a separate softmax. The general

formulation for a Seq2Seq network with Kullback-Leibler divergence is the following with true labels $Y = \{Y_t^q\}_t$:

$$\min_{\theta} E_{(X,Y)} \sum_{q=1}^{Q} \sum_{t=1}^{T} KL(Y_t^q || f_t^q(X;\theta;f_{i<t})).$$

$$\text{Here} \quad f_{i<t} = (f_{\bar{t}}^q)_{\bar{t}=1,q=1}^{t-1,Q}.$$

A general Seq2Seq network is effective for multiple time series problems that rely on a known training output size which is often not the case. However, if the natural sequence size is not known a priori, a basic Seq2Seq is insufficient. We create a model that is able to learn from training data that contains a capped output length during the training phase and predicts a dynamic output sequence during the prediction phase.

4.4.1. Dynamic Prediction Length Sequence to Sequence

To create a model that predicts a dynamic output length for each time series, we introduce a function, $g(f_{i\leq t})$ that captures confidence of the prediction at time t. Note that the confidence function may be dependent on previous values $f_1, f_2, ...f_t$. To determine output length, we measure the confidence function against a threshold value τ (which is a hyper-parameter) resulting in

$$\min_{\theta} E_{(X,Y)} \sum_{q=1}^{Q} \sum_{g(f_{i\leq t}^q(X;\theta;f_{i<t}))\geq\tau} KL(Y_t^q || f_t^q(X;\theta;f_{i<t})). \tag{4.1}$$

For the remainder of our work, we let $L_t^q := KL(Y_t^q || f_t^q(X;\theta;f_{i<t}))$ and $G_t^q := g(f_{i\leq t}^q(X;\theta;f_{i<t}))$. Note that we use the output of the softmax function $f_t^q(X;\theta;f_i)$ as a loss measure in the Kullback-Leibler divergence and the confidence measure for dynamic

output length. As an example, G_t^q can return the maximum predicted probability at time t, but it can also be total variation between one time step and the next.

The baseline loss function above contains a flaw. Since the set of optimal values are $\Theta^* := \{\theta^* : G_t^q(\theta^*) \geq \tau\}$, the model does not have to make accurate predictions to minimize the loss function. It can for example select θ such that $G_1^q < \tau$ and thus an "empty" sum. Therefore, we add a penalty to the loss function that punishes the model for not meeting the threshold to generate accurate prediction outputs.

Next, we introduce a second hyperparameter, λ, with the new penalty function in the loss. If the model makes few predictions, we can increase λ and decrease τ. To balance the F1 score and output length, we adjust each of these hyperparameters. By using the indicator function $I[G_t^q \geq \tau]$ for 1 if $G_t^q \geq \tau$ and 0 otherwise, the loss function reads

$$\min_{\theta} E_{(X,Y))} \sum_{q=1}^{Q} \sum_{t=1}^{T} \big(I[G_t^q \geq \tau] L_t^q + \lambda \max(\tau - G_t^q, 0) \big). \tag{4.2}$$

By design, if the model masks the Kullback-Leibler divergence (i.e. $G_t^q < \tau$), the penalty function produces a nonnegative output and vice-versa.

In addition to designing our penalty function to counter the Kullback-Leibler error, we make the penalty function in the form of a rectifier unit. We do this since rectifier units are effective with neural networks for training and have a simple non-vanishing gradient. Since there is only a single gradient difference between this function and the Kullback-Leibler loss, we save a lot of computational effort.

We also consider a smooth masking function to improve differentiability. We propose a form of the sigmoid function. With the sigmoid function our loss reads

$$\min_{\theta} E_{(X,Y)} \sum_{q=1}^{Q} \sum_{t=1}^{T} \left[H_t^{k,\tau}(G_t^q) L_t^q + \lambda \max(\tau - G_t^q, 0) \right]. \tag{4.3}$$

where

$$H_t^{k,\tau}(x) = \frac{1}{1+e^{-k(\frac{x-\tau}{1-\tau})}}$$

for positive and possibly large hyperparameter k. Since the loss function in (4.3) is differentiable, standard back propagation can be used. On the other hand, (4.2) is not differentiable since the number of summation term depends on trainable parameters θ.

Up until this point, we have not considered the various possibilities for the function $g(f_{i\leq t}^q(X;\theta;f_{i<t})$. To properly balance accuracy and future predictions, we require that the function can express the confidence of the model and/or be an effective measure of volatility. We choose four different functions that range in both complexity and time dependency:

- Maximum: $\max_j f_{t,j}^q(X;\theta;f_{i<t})$
- Confidence Distance: $\max_j(f_{t,j}^q(X;\theta;f_{i<t})) - \max_{k,k\neq j^*}(f_{t,k}^q(X;\theta;f_{i<t}))$ where $j^* := \arg\max_j(f_{t,j}^q(X;\theta;f_{i<t}))$
- Total Variation: $\max_j |f_{t,j}^q(X;\theta;f_{i<t}) - f_{t-1,j}^q(X;\theta;f_{i<t-1})|$
- Wasserstein/Earth Mover Distance: $W^1(f_t^q(X;\theta;f_{i<t}), f_{t-1}^q(X;\theta;f_{i<t-1}))$.

The first two functions are based upon the current prediction time while the other two are dependent on the current and previous outputs. Confidence distance and maximum measure confidence of the current prediction. If the model is struggling between two different labels (uncertainty), then we expect this value to be small enough to be below the threshold. Both Total Variation and the Wasserstein Distance are classic measures for the distance between two

probability distributions. Note that we specifically choose the first Wasserstein Distance, which is also known as the Earth Mover Distance. In addition, because total variation and the earth mover distance measure volatility, we define G_t^q to be the negative of these two measures, which then leads to $G_t^q \leq \tau$.

4.4.2. Training and Inference

There are differences between the training of a typical Seq2Seq model and our new model during training and prediction. For example, when our network stops at $\hat{T}(q)$ for time series q, we no longer consider the accuracy error terms for $t > \hat{T}(q)$ when computing the gradient.

In the forward pass, we stop when $G_{\bar{t}}^q < \tau$ occurs the first time for time series q and thus $\bar{t} = \bar{t}(q)$. It may happen that for some $G_{\hat{t}(q)}^q \geq \tau$ for some $\hat{t}(q) > \bar{t}(q)$. When this occurs, we do not consider the terms corresponding to $\hat{t}(q)$ in the Kullback-Leibler divergence or in our F1 score calculation. With the indicator function, the outputs from the Kullback-Leibler divergence are completely masked to zero. However, with the sigmoid function, the outputs below τ are only partially masked. In addition, when utilizing the sigmoid function, we calculate the F1 scores in the same manner as in the case of the indicator function.

After obtaining $\bar{t}(q)$ in the forward pass for each time series q, we change the loss to the following when using (4.1):

$$\min_{\theta} E_{(X,Y))} \sum_{q=1}^{Q} \left[\sum_{t=1}^{\bar{t}(q)} L_t^q + \sum_{\hat{t}=\bar{t}(q)+1}^{T} \lambda \max(\tau - G_{\hat{t}}^q, 0) \right].$$

From this point on, we do standard backpropagation treating $\bar{t}(q)$ as fixed for the current sample.

4.5. Computational Study

4.5.1. Data Preparation

We study our model on financial time series data. Our data consists of fourteen years of securities at five minute tick intervals for two sets consisting of twenty-two ETF's and five distinct commodities. The identity of the securities are unknown; therefore, we do not incorporate any additional features such as news and market announcements. To clarify, the dataset consists of only the single tick price rather than prices on multiple levels of an order book. In addition, we do not have trade volume information. The prices for the twenty-two ETF dataset are the returns of each security, which is the relative change in price from one 5 minute interval to the next. The five commodity dataset consists of the price of each security at each time t. In order to have consistency between the two datasets, we choose to only consider the returns of the five commodity dataset.

We create two representations of our data: one for the feed forward and seq2seq networks and the other for the convolutional seq2seq network. First, we create a sequence of inputs consisting of $\bar{T}$ returns for each financial asset. Since our data can be viewed as a continuous streaming sequence (no obvious beginning or end), we create sequences of various size $\bar{T}$. For each sequence, the next sequence moves forward by a 5 minute interval. We found that including this overlap increased prediction accuracy. In our best performing feed forward networks, our feature vectors consist of ten returns. For a seq2seq network we increase the sequence size to twenty.

Since an increase in the sequence size greatly increases the computation time of a seq2seq neural network, we consider a 1-dimensional convolutional LSTM. A convolutional LSTM

layer can take much larger sequences because the convolutional step in each LSTM node reduces the total number of sequences via a convolutional kernel. Since we input multiple returns from different financial assets, each channel of convolution represents a difference financial security (similar to RGB). We tried two models: convolution feeding into LSTM and ConvLSTM (Xingjian, Chen, Wang, Yeung, Wong, and Woo (2015)). All seq2seq models are embedded with our loss function unless stated otherwise.

For normalization, we use the standard approach of calculating the mean and standard deviation of the training set and using that to normalize both the training and validation/test sets. For our data, we use one year of training data (47,000 samples), and classify the next week of returns (1,000 samples).

We utilize a walk-forward methodology to evaluate our model over the entire dataset. After each training phase we move the training and testing windows one week forward. The first week of the previous training data is dropped form the training data of the next phase. Each training is warm-started (pretrained) from the previous window.

To classify the returns of each security, we split the labels into five classes: large upward movement, small upward movement, insignificant movement, small downward movement, and large downward movement.

We calculate the mean (μ_D) and standard deviation (σ_D) of the returns s_D of the previous day to create our labeling scheme of five classes for the returns of day $D+1$. To classify the return x_t^{D+1}on day $D+1$ at time t, we use the following rules.

$$x_t^{D+1} < \mu_D - \sigma_D,$$

$$\mu_D - \sigma_D \leq x_t^{D+1} < \mu_D - \beta\sigma_D,$$

$$\mu_D - \beta\sigma_D \leq x_t^{D+1} < \mu_D + \beta\sigma_D,$$

$$\mu_D + \beta\sigma_D \leq x_t^{D+1} < \mu_D + \sigma_D,$$

$$\mu_D + \sigma_D \leq x_t^{D+1}$$

Here β is a paramter. We set β such that roughly 50% of all values lie within insignificant movement ($\beta = 0.14$) for the twenty-two security dataset. For the other dataset, a few of the securities contain large data imbalances at the large upward and downward movements. Since this is the case, we pick $\beta = 0.1$ so that the majority class contains roughly 50% of all labels. These choices lead to class imbalances of roughly [5%, 20%, 50%, 20%, 5%].

4.5.2. Results

The following results are based upon the best sequence sizes, neurons per layer, optimization method, and number of layers found through hyperparameterization of each network. The models were trained on Titan X's and Nvidia 1080's and implemented in tensorflow. For each dataset, we have one model to classify all twenty-two ETF's from one dataset and one model to classify all five commodities in the other dataset. We test the ETF dataset on all baseline and final model architectures. Due to computational time constraints, we only calculate the results of the commodity dataset using the best model from the ETF dataset.

4.5.3. Baseline Test

For our first experiments, we use a FFN, LSTM network, LSTM Seq2Seq network, and a ConvLSTM Seq2Seq network in the setting of a fixed prediction of 1 or 10. The FFN consists of two layers with sixty-four neurons and ten returns for each security. We train the FFN with the

Table 4.1. F1 Scores of Baseline Models

Architecture	ETF's	Commodities
FFN (One Pred)	0.176	0.159
LSTM (One Pred)	0.209	0.286
ConvLSTM (One Pred)	0.513	0.410
LSTM Seq2Seq (One Pred)	0.598	0.513
LSTM Seq2Seq (Ten Pred)	0.509	0.474

stochastic gradient optimization method. The basic recurrent LSTM network consists of two LSTM layers, each with sixty-four neurons and input sequence size of twenty. For all recurrent networks, we use the ADAM optimization method by Kingma and Ba (2014), which is known to perform well for recurrent networks. We train for a number of epochs until the F1 score no longer increases on the validation dataset. Last, the sequence to sequence network consists of one encoder and one decoder layer, each with sixty-four neurons and either LSTM or ConvLSTM layers. We tried using deeper networks, but those resulted in much lower F1 score. We utilize the same model architecture for both the ETF and commodity datasets.

We are using financial security datasets, which typically contain imbalanced classes. When considering imbalanced classes, the traditional accuracy measure does not display the true effectiveness of a model. Therefore, we utilize the F1 score, which accounts for imbalanced classes. The F1 score is calculated via a one-against-all structure of precision and recall for each of the five classes. The values we report are an average of the F1 scores for each class and security.

The FFN, LSTM, LSTM Seq2Seq, and ConvLSTM Seq2Seq results are presented in Table 4.1. To produce these results, we give each model a warm start by training on 5 windows of data. We then train the models for 25 more windows on the 22 ETF's and 5 commodities datasets. The F1 scores in Table 4.1 are the average test scores over this 25 window period.

We use our baseline results in Table 4.1 to determine the best architecture for our dynamic output length model and to examine the differences of the various seq2seq networks. We observe that each single prediction model generally increases in performance when using more robust models (such as seq2seq). As expected, the seq2seq models and recurrent LSTM models far outperform the FFN, but we did not expect such a large performance increase when utilizing a seq2seq model. Even when making 10 predictions forward in time, the seq2seq model more than doubles the F1 score compared to the traditional LSTM network that makes a single prediction. With ConvLSTM layers, we found that the best formulation is utilizing 1D convolutions with each security representing a channel in the image. Ultimately, we find that the best ConvLSTM seq2seq model does not perform as well as the LSTM seq2seq model. Since the LSTM seq2seq model is by far the most accurate, we choose it for our dynamic output length model.

4.5.4. Output Length and Accuracy

We begin by examining the behavior of our dynamic model with the confidence distance (CD) threshold function, which turns our to be the best one as shown below. We want the model to find the best number of predictions for each security and each input. To verify this behavior, we train the seq2seq LSTM model with parameters $\tau = 0.4$ and $\lambda = 1.0$ on the first training window. We utilize these particular hyperparameter values since they showcase dynamic prediction length and result in superior performance. This pair and other hyperparameter pairs are visualized in the next section.

In Figure 4.1, we present the training F1 scores and average output length for a single test window of the 22 ETF's dataset. In the image on the left we observe that the validation loss

decreases for roughly 20 epochs before it begins to overfit. During our model training, we use the model weights that result in the best validation F1 score (which coincides with the best lowest validation loss) as a starting point for the next training window. For the output length image on the right, the output length margin between training and validation is small at the beginning of training. However, as the model continues to train, the average number of predictions for training and validation diverges slightly. Also, we observe that as the model continues to train, the number of predictions increases to a limit at around 20 epochs, which matches the loss.

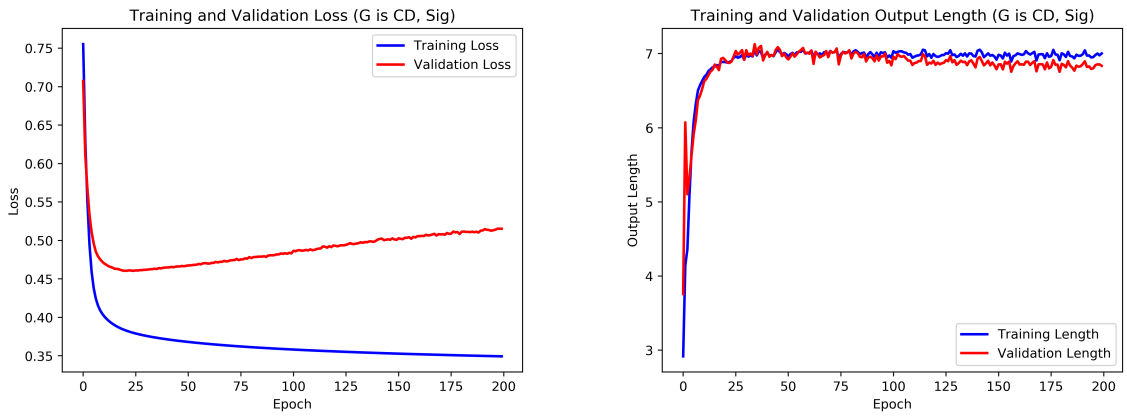

Figure 4.1. Total loss (KL divergence plus max) and average output length for a single window with the CD thresholding function.

One particularly interesting behavior we find with the dynamic model is that each security had variable prediction lengths. In Figure 4.2 we present an example of this behavior under the same setting. The prediction lengths are an average of 5 test set windows after a 5 window warm-up period with the 22 ETF's dataset. We notice that the number of predictions varies from roughly 5 predictions to nearly 10. In addition to the the average number of predictions, we place the majority class label percentage, which is a proxy to volatility, of each security at the

top of each corresponding bar. There does not appear to be a correlation between volatility and the average prediction length. We hypothesize that the difference in prediction length is due to the difference in volatility of the training and test sets of each security. Financial securities tend to have covariate shift, and our model may capture this during inference, which we examine in Figure 4.3.

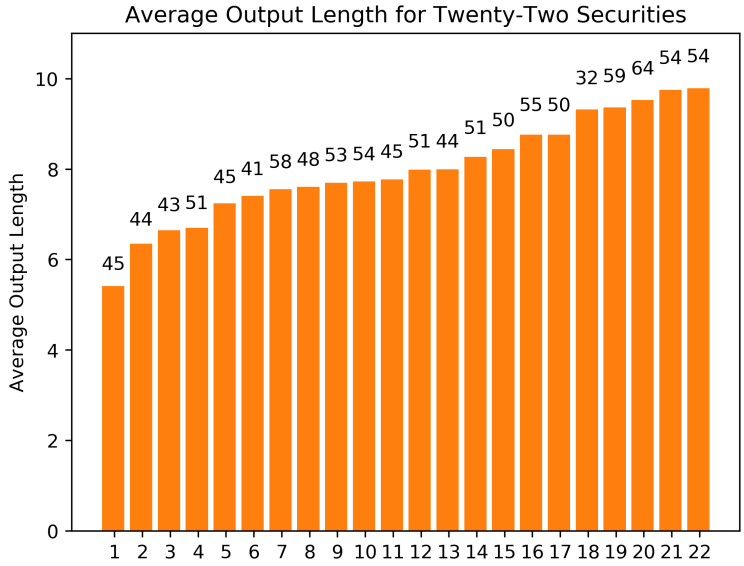

Figure 4.2. Average number of predictions for each ETF over a 5 window period. At the top of each bar is the average percentage of labels for the majority class for the same 5 window period.

To test the variance, we perform the standard t-test between the samples of the training and test sets. Specifically, we test the difference between the means of final input (the actual return) of each training and test sequence. We expect that less variance between the training and test sets (a larger p-value) correlates with a longer output length. In Figure 4.3, we use the same 5 windows, threshold function, and values for τ and λ as in Figure 4.2. Each point in the plot

corresponds to the average p-value of the t-test between training and test sets, and the average prediction length for each of the 22 ETF's.

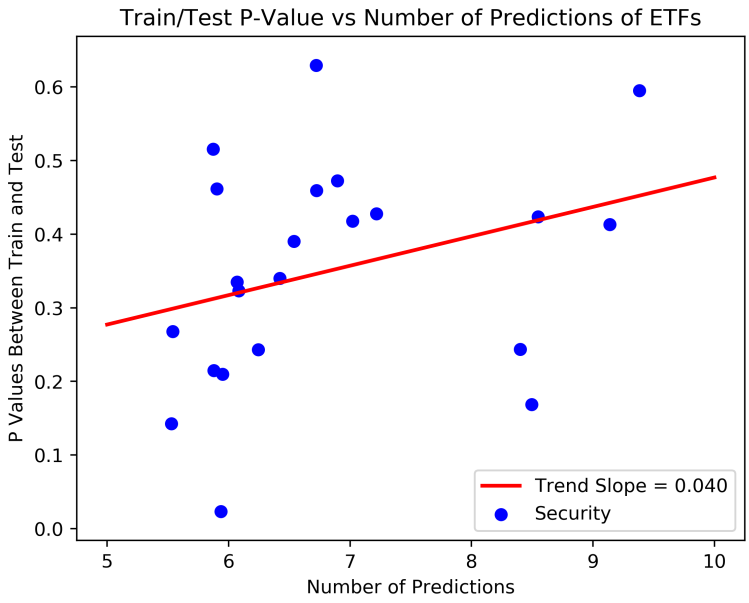

Figure 4.3. For each ETF we calculate the p-value between the training and test sets using the standard t-test. We compare this to the corresponding output length of the test set.

Next we examine the benefits of a dynamic output length. In the following two sections, we explore the effectiveness of our four proposed confidence functions.

4.5.5. Maximum and Confidence Distance Confidence Functions

We begin with the two confidence functions that depend only upon the current prediction: maximum and CD. We study the sensitivity of τ and λ, and the comparison between the indicator (Ind) and sigmoid (Sig) functions, i.e. loss functions (4.2) and (4.3) respectively. In the following sections, we refer to the indicator and sigmoid functions as masking functions because they mask the output from the Kullback-Leibler divergence.

A large gradient for $\lambda \max(\tau - G_t^q, 0)$ means that we encourage the model to make more predictions. We can control the output length by adjusting the hyperparameters λ and τ. In Figure 4.4 we present a range of τ and λ values and the respective F1 scores with G_t^q being the maximum. In addition, we place an annotation of the best pair (τ, λ) that results in the largest distance above the static curve in bold. We add this annotation to all figures of this type.

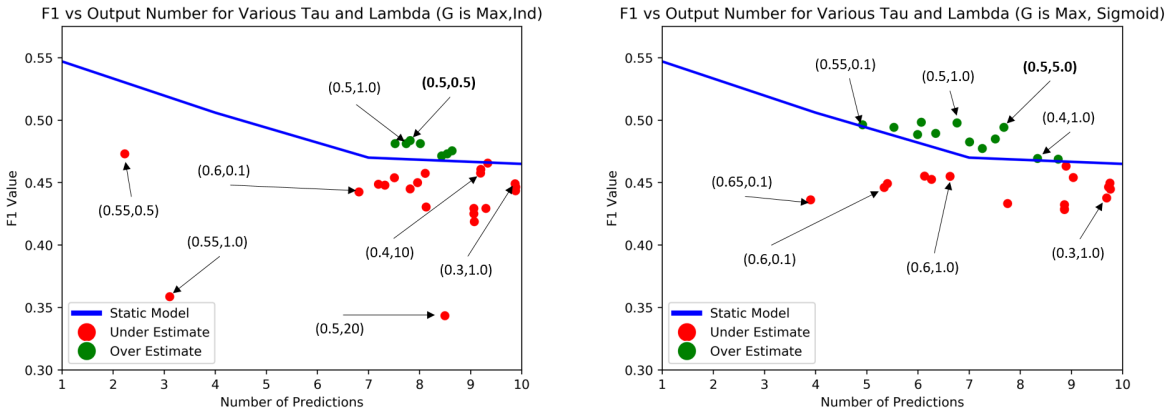

Figure 4.4. F1 and average output length for pairs of τ and λ. We use the maximum confidence function with the indicator function on the left and sigmoid on the right.

In Figure 4.4, we first create a line that gives the F1 scores from a static seq2seq model for 5 training windows after a 5 window warm start. We compute F1 for $1, 4, 7, 10$ predictions by the static model and interpolate the remaining values. It is important to point out that the F1 scores here are different than in Table 4.1 since we are measuring 5 windows rather than 25. We expect the dynamic model to be above this curve when using appropriate values for (τ, λ). Each point in Figure 4.4 is the average output length and F1 score of the 22 ETF dataset for a pair (τ, λ). On the left is the indicator masking function and the right is the sigmoid masking function. For both figures, we use the same values for τ and λ. We find that with $\tau = 0.5$, our model has high accuracy with a relatively large average output length.

The red points are a pair (τ, λ) that perform worse than the static model and the green are those that perform better. We expect to find many pairs of hyperparameters that lead to superior F1 scores since our model uses a dynamic prediction length for each sample. There are a few outliers that fall well below the estimated performance in the figure on the left, which uses the indicator masking function. The sigmoid function on the right has many more points above the static curve and does not have any points below the 0.4 F1 score. We observe that many red points usually have large λ's. When λ is too big, the model stops training for prediction accuracy and only focuses on creating more predictions. Note that not all large λ's lead to strictly poor performance since the best sigmoid pair has $\lambda = 5.0$.

Next we consider CD. As in Figure 4.4, Figure 4.5 shows a hyperparameter search (τ, λ) with the indicator masking function on the left and sigmoid masking function on the right. Note that with CD, the number of points below the static line is smaller compared to the maximum function. When λ is too large, the indicator version has three points that fall far below our estimation. It is important to point out that there is a large distance between the green points and the blue curve that is rarely seen with other confidence functions. Based on these results, we recommend using the CD over the maximum function. It requires very little extra computation and leads to better F1 with a similar average output length.

In Figures 4.4 and 4.5, we show that large λ's usually lead to inferior models. Therefore, we explore τ to see if we can find a relationship between this hyperparameter and the average output length. We present two images in Figure 4.6 of the sensitivity of the indicator and sigmoid masking functions with maximum and CD functions. We create these figures by choosing $\lambda = 0.1$, which produces the best performance for all cases of τ and plot the relationship between

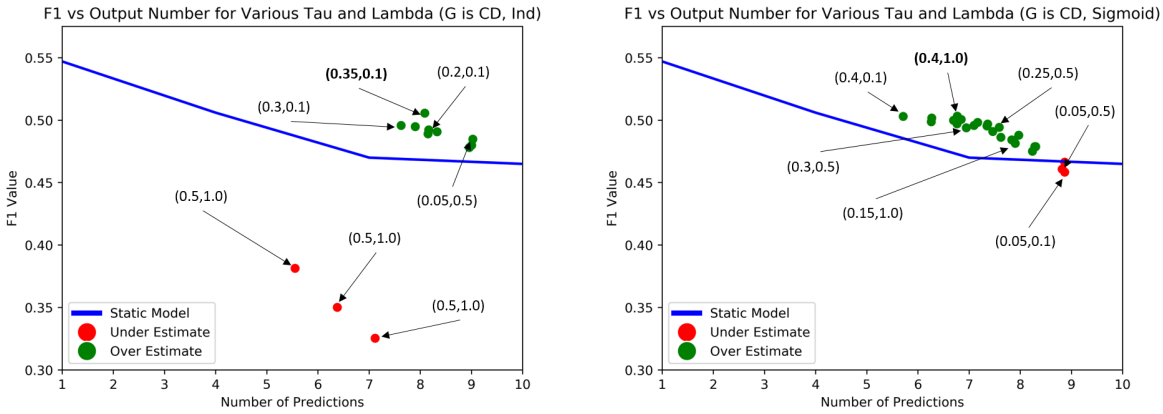

Figure 4.5. F1 score and average output length for pairs of τ and λ. We use CD with the indicator function on the left and sigmoid function on the right.

average number of predictions and τ. Note that we use the same approach of utilizing a 5 window warm start and measure the average output length based upon the next 5 windows.

On the left, we observe that the slope for the sigmoid masking function is sharper than the indcator function for both confidence functions. In the case of these two functions, the sensitivity is not nearly as large as the other two confidence functions that we explore in the next section. For CD, the slope is much smaller (in terms of absolute value), which provides additional evidence to recommend it over the maximum confidence function.

Given these two confidence and masking functions, we recommend using CD with the sigmoid masking function. Although the sigmoid masking function is more sensitive to τ, we find that sigmoid does not have as many points falling under the static curve. Next, we examine the two confidence functions that depend on the current and previous predictions.

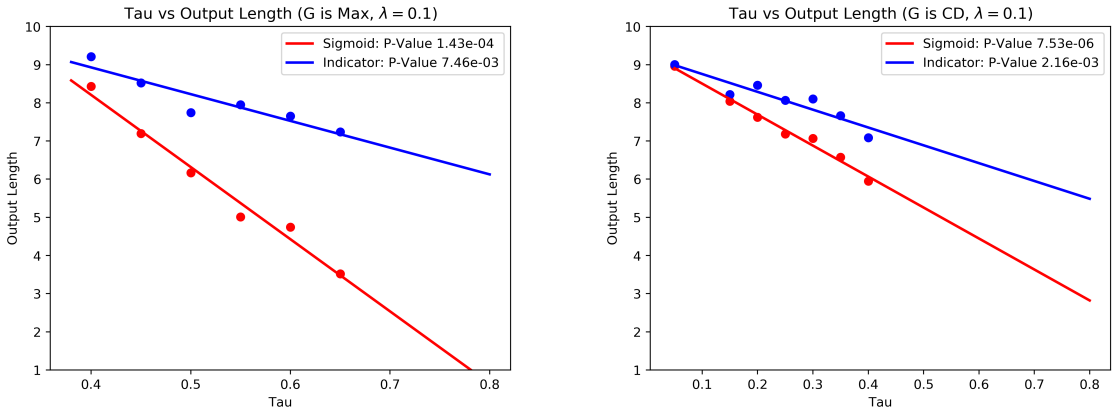

Figure 4.6. The output length and τ values for both CD and maximum functions with $\lambda = 0.1$. The trend lines are made via linear regression and show high statistical significance between τ and output length.

4.5.6. Total Variation and Wasserstein Confidence Functions

The other two functions we test are total variation (TV) and the first Wasserstein distance (EMD). We design the first two functions, maximum and CD, to find the confidence of the current prediction using only the current prediction and input sequence. On the other hand, we use TV and EMD to determine the volatility between the current and previous prediction. If the volatility is low, we expect the model to be confident enough to make several predictions. We first examine the results from TV in Figure 4.7.

TV does not garner high performance with either masking function. There are some points with the sigmoid masking function that are slightly better than the expected F1 scores, but TV is clearly not a successful measure. There are likely some cases where the TV may be large between two predictions, but the model may still be confident about that prediction. For example, if the prediction changes from one label to another at the next prediction step, the TV between these two predictions may be large. However, this does not necessarily imply that the

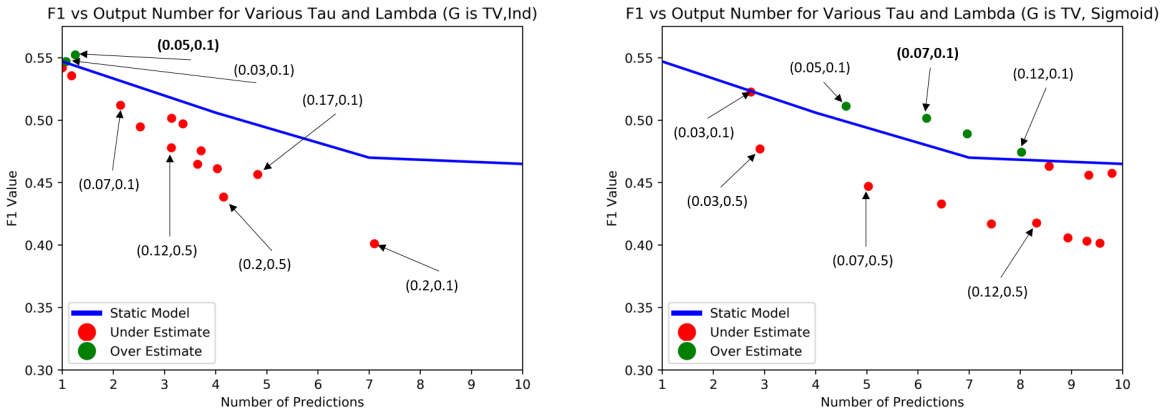

Figure 4.7. The F1 score and respective output length for various (τ, λ) pairs with TV. The indicator function is on the left while the sigmoid function is on the right.

model is not confident about its next prediction. Also note that there are two rows of points above and below the static curve of the sigmoid version. The difference between these two is that the value for λ is larger for the points below the estimated curve, which is similar to the results we obtain form the maximum and CD functions.

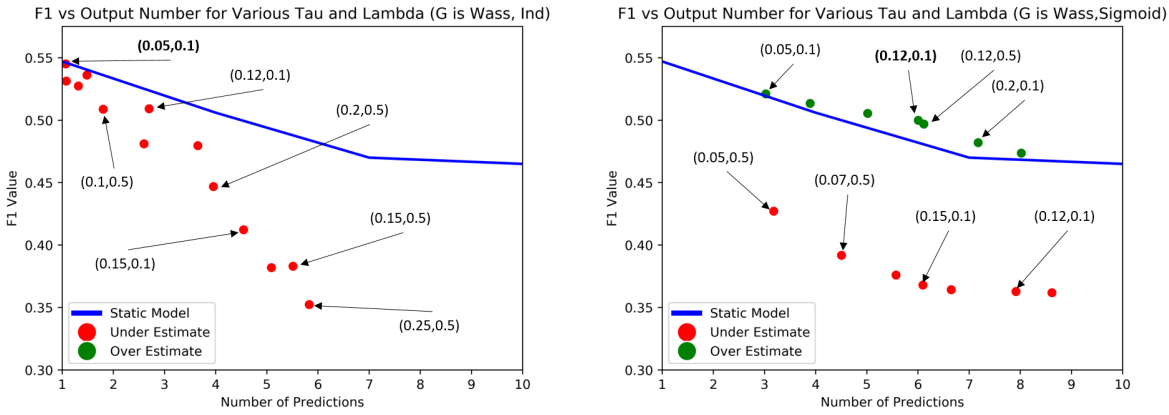

Figure 4.8. The F1 score and respective output length for various (τ, λ) pairs with EMD. The indicator function is on the left while the sigmoid function is on the right.

In Figure 4.8, we observe that EMD contains a similar pattern to TV. Similar to TV, the indicator masking function with EMD performs poorly on the left compared to the better F1 scores of the sigmoid masking function on the right. EMD performs slightly better than TV with a few more points above the estimated curve. This likely occurs because EMD is a more robust probability measure. Instead of finding the maximum distance between two probability measures P and Q (TV), EMD finds the cost of transforming the entirety of one distribution P into another distribution Q. EMD is more robust, but a label change from one time step to the next may correlate with a large EMD. However, this does not always imply lower prediction confidence.

To conclude our examination of TV and EMD, we observe the relationship between the number of predictions and τ in Figure 4.9. Note that the relationship is the opposite of the maximum and CD functions since we reverse the relationship with respect to τ, which we clarify in Section 4.3.1.

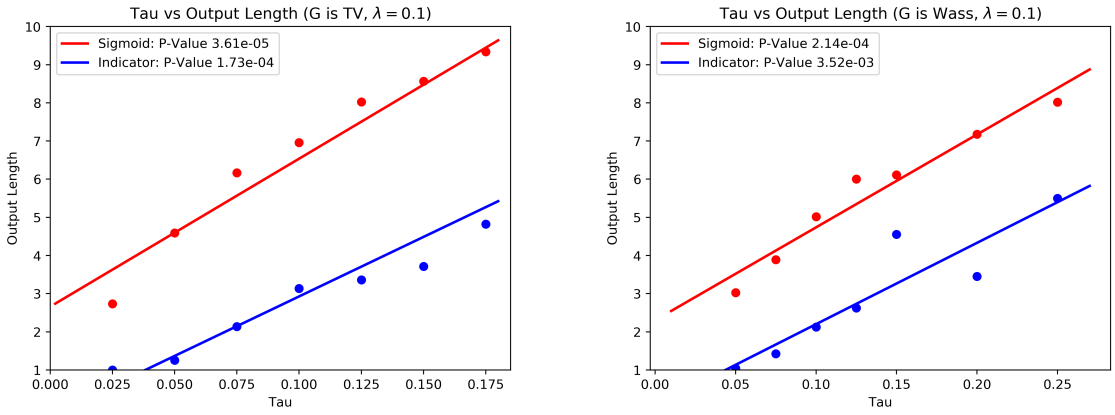

Figure 4.9. The output length and τ values for both TV and the EMD with $\lambda = 0.1$. The trend lines are made via traditional linear regression and show high statistical significance between τ and output length.

The slopes for the sigmoid versions of both metrics are larger than their indicator function counterparts. We also observe that the slopes are large overall when compared to both the maximum and CD confidence functions. Slight changes in τ lead to large changes in output lengths for TV and EMD. In general, we found that the values of these two metrics tend to be small, which is what leads to their sensitivity to τ. Overall, for all metrics, the sigmoid masking function slope is larger in magnitude than the same version with the indicator masking function. Also, TV and EMD show that confidence in predictions depends upon more than volatility.

4.5.7. Five Commodity Results

Next, we test our model on the five commodity dataset. Since we observe that the CD with the sigmoid masking function provide the best performing model with the ETF dataset, we use the identical functions on the commodity dataset. In addition, we create four static models predicting $1, 4, 7, 10$ time steps each to create the blue curve as with the previous ETF figures. In Figure 4.10, we present the dynamic model results with the commodity dataset. As with the previous figures of the same type, we run the model for a total of 10 walk-forward steps and average the final 5 F1 scores, which are reported in Figure 4.10.

The first thing we observe is that this is the first time that every single pair (τ, λ) is above the static predictions. This probably occurs because of volatility in the commodity dataset. Some commodities are extremely volatile, and the price ranges from big highs to lows for a high percentage of the labels. In addition, other commodities in the dataset have very low volatility for a long period of time. Because of the skewed nature of the commodity dataset, it is possible for our model to only choose to predict at times when it can make many correct predictions.

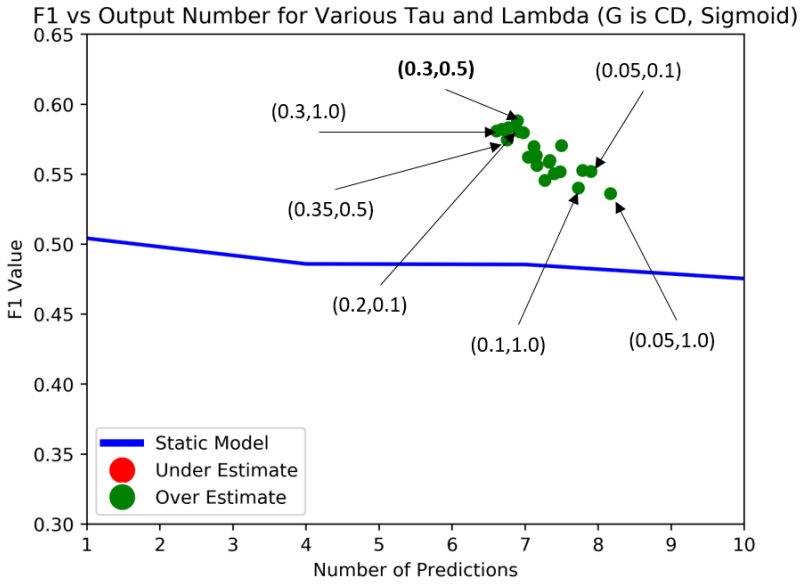

Figure 4.10. The F1 score and respective output length for various (τ, λ) pairs with CD and the sigmoid masking function with the five commodity dataset.

To observe the dynamic model in Figure 4.11, we observe the F1 scores of our best dynamic commodity model and two static models over 10 walk forward periods.

The F1 score of our dynamic model is superior at nearly every walk forward time period in comparison to a static single prediction model, except for walk forward period 5. As expected, the single prediction static model has a better F1 score on average than the static model making 7 predictions. However, the difference between 1 prediction and 7 predictions is small, which is in stark contrast to the ETF dataset. This provides additional evidence that the price volatility of the labels makes a static number of predictions difficult, and shows that a dynamic model can be advantageous with a time series dataset that has a skewed distribution.

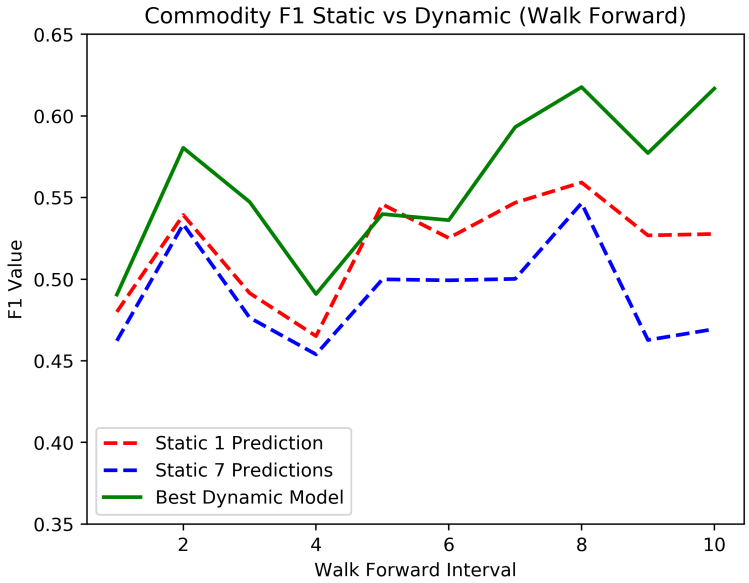

Figure 4.11. The F1 score of our best dynamice model: $\tau = 0.3$ and $\lambda = 0.1$ and an average prediction length of 7. We showcase its F1 score for each walk forward period against two static models.

4.5.8. Summary of Results

In Table 4.2, we present a summary of the best results for our dynamic model. To measure which pair (τ, λ) is best, we measure the relative improvement between the F1 score of the dynamic model and the equivalent prediction length F1 score of the static model (from the blue curves shown in previous figures). The dynamic model with CD and sigmoid does extremely well for the commodity dataset. The same functions also give the best F1 score with the ETF dataset.

Table 4.2. Dynamic Model Summary (All Sigmoid)

Architecture	F1 Gap %	(τ, λ)	Prediction Length
Dynamic Max (ETF)	5.44	$(0.50, 5.0)$	7.68
Dynamic CD (ETF)	6.45	$(0.40, 1.0)$	6.76
Dynamic TV (ETF)	4.49	$(0.07, 0.1)$	6.16
Dynamic EMD (ETF)	3.72	$(0.12, 0.1)$	6.00
Dynamic CD (Commodity)	21.2	$(0.30, 0.5)$	6.89

4.6. Summary

In this work, we create a new loss function with the seq2seq network that makes a dynamic number of predictions for each input sequence. In addition, we construct a new metric called "Confidence Distance" to measure the confidence the model has when making a prediction. When testing the new model, we find that a dynamic prediction length model can outperform a similar static seq2seq network. In addition, we find that of our four confidence functions, CD gives the most accurate predictions over maximum, TV, and the EMD. We examine two versions of our model with each confidence function: masking the Kullback-Leibler divergence with the indicator function and masking with the sigmoid function. We find that the sigmoid function leads to better and more reliable prediction F1 even though it is more sensitive to hyperparameter τ.

When using the dynamic model, we recommend to keep the value for λ low and to vary τ to get the desired output length to F1 ratio. Seq2Seq models perform well with financial security data, and we recommend their use and to make the first decoder input the label associated with

the final encoder input. For the best dynamic output length performance, the best setting utilizes $\tau \in [0.05, 0.5]$ and $\lambda \leq 1.0$ with the confidence distance metric and sigmoid masking.

CHAPTER 5

Conclusion

In the first chapter of our work, we introduced the three basic neural network architectures: feed forward, convolutional, and recurrent neural networks.

In Chapter 2, we explored an adversarial multiagent problem in the form of NBA basketball. This dataset consisted of trajectory data gathered from cameras installed on courts by STATS LLC. The goal of the work was to predict the probability that a shot is made given that a shot is taken. We introduced faded trajectories to represent player movement over time on the court. We found that using convolutional neural networks on the faded trajectories combined with classic features crafted for a feed forward network produced a model that is over 60% accurate at predicting whether a shot will be made.

In Chapter 3, we challenged the notion of having a single activation function per neuron. We created a new architecture that can be applied to any type of neural network. In addition to the typical activation functions used in deep learning, we introduced a unique set of activations and found that this set is superior for the datasets MNIST, ISOLET, CIFAR-10, CIFAR-100, and STL-10. In addition, we found that with the typical set of activation functions used in neural networks, that the best activation function for a given problem depends on layer depth, network architecture, and on the dataset itself.

In Chapter 4, we focused on a specific type of recurrent network, the Sequence to Sequence Network. We created a novel model that can generate a dynamic number of predictions based upon sequence data. To create our new model, we introduced a new term to the loss function and

developed a confidence function to determine the confidence of each prediction. We showcased our model with two financial security datasets, which consisted of security returns at five minute intervals. We found that our model can succesfully make a dynamic number of predictions and that our new confidence metric, called the confidence distance, gave the highest F1 score at each prediction length. Finally, we found that a model that can make a dynamic number of predictions can achieve superior performance when compared to models restricted to a fixed output length.

References

Agostinelli, F., M. Hoffman, P. Sadowski, and P. Baldi (2015): "Learning activation functions to improve deep neural networks," in *International Conference on Learning Representations*.

Akita, R., A. Yoshihara, T. Matsubara, and K. Uehara (2016): "Deep learning for stock prediction using numerical and textual information," in *15th International Conference on Computer and Information Science*, 1–6.

Arjovsky, M., S. Chintala, and L. Bottou (2017): "Wasserstein GAN," *arXiv preprint arXiv:1701.07875*.

Bahdanau, D., K. Cho, and Y. Bengio (2014): "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*.

Bao, W., J. Yue, and Y. Rao (2017): "A deep learning framework for financial time series using stacked autoencoders and long-short term memory," *PLOS ONE*.

Borovykh, A., S. Bohte, and C. W. Oosterlee (2017): "Conditional time series forecasting with convolutional neural networks," *arXiv preprint arXiv:1703.04691*.

Cervone, D., A. D'Amour, L. Bornn, and K. Goldsberry (2016): "A multiresolution stochastic process model for predicting basketball possession outcomes," *Journal of the American Statistical Association*, 111, 585–599.

Chen, J. (2016): "Combinatorially generated piecewise activation functions," *arXiv preprint arXiv:1605.05216*.

Chen, K., Y. Zhou, and F. Dai (2015): "A LSTM-based method for stock returns prediction: A case study of china stock market," in *IEEE International Conference on Big Data*, 2823–2824.

Chong, E., C. Han, and F. C. Park (2017): "Deep learning networks for stock market analysis and prediction: Methodology, data representations, and case studies," *Expert Systems with Applications*, 187–205.

Clevert, D.-A., T. Unterthiner, and S. Hochreiter (2016): "Fast and accurate deep network learning by exponential linear units (elus)," in *International Conference on Learning Representations*.

Ding, X., Y. Zhang, T. Liu, and J. Duan (2015): "Deep learning for event-driven stock prediction," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Dixon, M., D. Klabjan, and J. H. Bang (2016): "Classification-based financial markets prediction using deep neural networks," *Algorithmic Finance*, 1–11.

Dosovitskiy, A., J. T. Springenberg, M. Riedmiller, and T. Brox (2014): "Discriminative unsupervised feature learning with convolutional neural networks," in *Advances in Neural Information Processing Systems*, 766–774.

Erhan, D., Y. Bengio, A. Courville, and P. Vincent (2009): "Visualizing higher-layer features of a deep network," *University of Montreal*.

Franks, A., A. Miller, L. Bornn, and K. Goldsberry (2015a): "Counterpoints: Advanced defensive metrics for nba basketball," in *MIT Sloan Sports Analytics Conference, Boston, MA*.

Franks, A., A. Miller, L. Bornn, K. Goldsberry, et al. (2015b): "Characterizing the spatial structure of defensive skill in professional basketball," *The Annals of Applied Statistics*, 9, 94–121.

Frogner, C., C. Zhang, H. Mobahi, M. Araya, and T. A. Poggio (2015): "Learning with a wasserstein loss," in *Advances in Neural Information Processing Systems*, 2053–2061.

Glorot, X., A. Bordes, and Y. Bengio (2011): "Deep sparse rectifier neural networks." in *AISTATS*, volume 15, volume 15, 275.

Goodfellow, I., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio (2014): "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, 2672–2680.

Goodfellow, I. J., D. Warde-Farley, M. Mirza, A. C. Courville, and Y. Bengio (2013): "Maxout networks." in *Internation Conference on Machine Learning*, volume 28, volume 28, 1319–1327.

Graves, A. (2016): "Adaptive computation time for recurrent neural networks," *arXiv preprint arXiv:1603.08983*.

Graves, A., G. Wayne, and I. Danihelka (2014): "Neural turing machines," *arXiv preprint arXiv:1410.5401*.

Gulcehre, C., M. Moczulski, M. Denil, and Y. Bengio (2016a): "Noisy activation functions," in *International Conference on Machine Learning*, 3059–3068.

Gulcehre, C., M. Moczulski, F. Visin, and Y. Bengio (2016b): "Mollifying networks," *arXiv preprint arXiv:1608.04980*.

He, K., X. Zhang, S. Ren, and J. Sun (2016): "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 770–778.

Hochreiter, S. and J. Schmidhuber (1997): "Long short-term memory," *Neural computation*, 1735–1780.

Huang, K.-Y. and W.-L. Chang (2010): "A neural network method for prediction of 2006 world cup football game," in *Neural Networks, The International Joint Conference on*, IEEE, 1–8.

Ioffe, S. and C. Szegedy (2015): "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, 448–456.

Jaderberg, M., K. Simonyan, A. Zisserman, et al. (2015): "Spatial transformer networks," in *Advances in Neural Information Processing Systems*, 2017–2025.

Janocha, K., W. M. Czarnecki, et al. (2017): "On loss functions for deep neural networks in classification," *Schedae Informaticae*, 2016, 4959.

Jin, X., C. Xu, J. Feng, Y. Wei, J. Xiong, and S. Yan (2016): "Deep learning with s-shaped rectified linear activation units," in *AAAI Conference on Artificial Intelligence*.

Karpathy, A. and L. Fei-Fei (2015): "Deep visual-semantic alignments for generating image descriptions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 3128–3137.

Kingma, D. P. and J. Ba (2014): "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*.

Krauss, C., X. A. Do, and N. Huck (2017): "Deep neural networks, gradient-boosted trees, random forests: Statistical arbitrage on the s&p 500," *European Journal of Operational Research*, 689–702.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012): "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*.

Li, H., W. Ouyang, and X. Wang (2016): "Multi-bias non-linear activation in deep neural networks," in *International Conference on Machine Learning*.

Loeffelholz, B., E. Bednar, K. W. Bauer, et al. (2009): "Predicting nba games using neural networks," *Journal of Quantitative Analysis in Sports*, 5, 1–15.

Lucey, P., A. Bialkowski, P. Carr, S. Morgan, I. Matthews, and Y. Sheikh (2013): "Representing and discovering adversarial team behaviors using player roles," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2706–2713.

Lucey, P., A. Bialkowski, P. Carr, Y. Yue, and I. Matthews (2014): "How to get an open shot: analyzing team movement in basketball using tracking data," *MIT SSAC*.

Mao, J., W. Xu, Y. Yang, J. Wang, Z. Huang, and A. Yuille (2015): "Deep captioning with multimodal recurrent neural networks (m-rnn)," in *International Conference of Learning Recognition*.

McCabe, A. and J. Trevathan (2008): "Artificial intelligence in sports prediction," in *Information Technology: New Generations, 2008. Fifth International Conference on*, IEEE, 1194–1197.

Miller, A., L. Bornn, R. P. Adams, and K. Goldsberry (2014): "Factorized point process intensities: A spatial analysis of professional basketball." in *Internation Conference on Machine Learning*, 235–243.

Nair, V. and G. E. Hinton (2010): "Rectified linear units improve restricted boltzmann machines," in *International Conference on Machine Learning*, 807–814.

Nalisnick, E. (2014): "Predicting basketball shot outcomes," `https://enalisnick.wordpress.com/2014/11/24/predicting-basketball-shot-outcomes/`.

Niaki, S. T. A. and S. Hoseinzade (2013): "Forecasting s&p 500 index using artificial neural networks and design of experiments," *Journal of Industrial Engineering International*, 1.

Perše, M., M. Kristan, S. Kovačič, G. Vučkovič, and J. Perš (2009): "A trajectory-based analysis of coordinated team activity in a basketball game," *Computer Vision and Image Understanding*, 113, 612–621.

Scardapane, S., M. Scarpiniti, D. Comminiello, and A. Uncini (2016): "Learning activation functions from data using cubic spline interpolation," *arXiv preprint arXiv:1605.05509*.

Sirignano, J. A. (2016): "Deep learning for limit order books," *arXiv preprint arXiv:1601.01987*.

Socher, R., A. Karpathy, Q. V. Le, C. D. Manning, and A. Y. Ng (2014): "Grounded compositional semantics for finding and describing images with sentences," *Transactions of the Association for Computational Linguistics*, 2, 207–218.

Springenberg, J. T., A. Dosovitskiy, T. Brox, and M. Riedmiller (2014): "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*.

Srivastava, N., G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014): "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research*, 15, 1929–1958.

Srivastava, R. K., K. Greff, and J. Schmidhuber (2015): "Training very deep networks," in *Advances in Neural Information Processing Systems*, 2377–2385.

Sukhbaatar, S., J. Weston, R. Fergus, et al. (2015): "End-to-end memory networks," in *Advances in Neural Information Processing Systems*, 2440–2448.

Sutskever, I., O. Vinyals, and Q. V. Le (2014): "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems*, 3104–3112.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich (2015): "Going deeper with convolutions," in *Proceedings of the IEEE*

conference on computer vision and pattern recognition.

Trottier, L., P. Giguère, and B. Chaib-draa (2016): "Parametric exponential linear unit for deep convolutional neural networks," *arXiv preprint arXiv:1605.09332*.

Vinyals, O., M. Fortunato, and N. Jaitly (2015a): "Pointer networks," in *Advances in Neural Information Processing Systems*, 2692–2700.

Vinyals, O., A. Toshev, S. Bengio, and D. Erhan (2015b): "Show and tell: A neural image caption generator," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 3156–3164.

Wang, K.-c. and R. Zemel (2016): "classifying nba offensive plays using neural networks," in *MIT Sloan Sports Analytics Conference, Boston, MA*.

Wang, Z., A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli (2004): "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*.

Wei, X., L. Sha, P. Lucey, P. Carr, S. Sridharan, and I. Matthews (2015): "Predicting ball ownership in basketball from a monocular view using only player trajectories," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 63–70.

Weston, J., S. Chopra, and A. Bordes (2015): "Memory networks," in *International Conference on Learning Representations*.

Wickramaratna, K., M. Chen, S.-C. Chen, and M.-L. Shyu (2005): "Neural network based framework for goal event detection in soccer videos," in *Multimedia, IEEE International Symposium on*, IEEE, 8–pp.

Xingjian, S., Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, and W.-c. Woo (2015): "Convolutional LSTM network: A machine learning approach for precipitation nowcasting," in *Advances in Neural Information Processing Systems*, 802–810.

Xu, K., J. Ba, R. Kiros, K. Cho, A. C. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio (2015): "Show, attend and tell: Neural image caption generation with visual attention." in *ICML*, volume 14, volume 14, 77–81.

Yue, Y., P. Lucey, P. Carr, A. Bialkowski, and I. Matthews (2014): "Learning fine-grained spatial models for dynamic sports play prediction," in *Data Mining, 2014 IEEE International Conference on*, IEEE, 670–679.

Zeiler, M. D. (2012): "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*.

Zhou, Z.-H., J. Wu, and W. Tang (2002): "Ensembling neural networks: many could be better than all," *Artificial Intelligence*, 137, 239–263.